8-1991

A study of vector and parallel processing

Alan Arthur Luchuk

To the Graduate Council:

I am submitting herewith a thesis written by Alan Arthur Luchuk entitled "A study of vector and parallel processing." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald W. Bouldin, Major Professor

We have read this thesis and recommend its acceptance:

Robert Bodenheimer

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Alan Arthur Luchuk entitled *A Study of Vector and Parallel Processing*. I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald W. Bouldin

Donald W. Bouldin, Major Professor

We have read this thesis and recommend its acceptance:

[illegible]

[illegible]

Accepted for the Council:

[illegible]

Associate Vice Chancellor and
Dean of The Graduate School

A STUDY OF VECTOR AND PARALLEL PROCESSING

A Thesis Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Alan Arthur Luchuk
August 1991

Acknowledgements

I acknowledge the contributions of the many that have helped make this thesis a reality. My family, friends, thesis committee, supervisors and employer, professors, colleagues, the University of Tennessee Computing Center, the Cornell National Supercomputing Facility, the Computational Mechanics Corporation, and others have encouraged and supported its completion. I especially acknowledge the contributions of Dr. Margaret Mason of the University of Tennessee Computing Center and Mr. Joe Orzechowski of the Computational Mechanics Corporation. Their encouragement and support contributed greatly to the timely completion of my research and write-up. Most of all, I acknowledge my Creator. Without His blessings and grace, none of this would have been possible.

"I can do everything through him who gives me strength." -- Phillipians 4:13, NIV

Abstract

This document is a thesis for a Master of Science degree in Electrical Engineering. This thesis explains the relevance of vector processing and parallel processing, the program candidate selection procedures, and my vector-enabling and parallel-enabling procedures, experiences, and results. This thesis also provides supplemental information about the supercomputing facilities available to researchers at the University of Tennessee, the target supercomputer (the IBM 3090), how vector and parallel processing work, the available software tools, and Amdahl's Law.

My research focused on two programs: AKCESS and a benchmark program. AKCESS is a set of programs that form a general-purpose finite element modeling tool. I vector-enabled its single-program computational engine. I developed the benchmark program to exploit vector and parallel processing. With it, I estimated the IBM 3090's scalar, vector, and parallel computation rates.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Scientists Find Supercomputers Useful

As the depth and breadth of scientific research increases, scientists often cannot experimentally find or verify new research results. The repeatability, difficulty, or cost of an experiment prohibits its execution. For example, experiments to verify nuclear winter theories cannot be repeated. Studying plasma effects occurring within the sun exemplifies other difficult research. One readily can find many other examples of difficult or impossible experiments.

Rather than ignore difficult or impossible experiments, scientists mathematically model them. These models produce equation systems that the scientists may solve via analytical or numerical methods. Often, the analytic solutions either do not exist or are difficult to obtain. Consequently, the scientists solve the equation systems via numerical methods. These numerical methods can require intractably large numbers of calculations to produce a precise solution. To make the solutions computationally tractable, the scientists simplify the models.

The execution time required to solve a model numerically constrains the model's computational tractability on a given computer. For example, if the model cannot be solved between the times a computer must be shut down for maintenance, it is computationally intractable on that computer. Also, if forecasting models (e.g., weather models) cannot be solved before the events occur, the solution is useless. Finally, if the results cost less to get experimentally than numerically, budget managers would prefer that the scientists get the results experimentally.

Because supercomputers can do more calculations in a given amount of time than less-capable computers can, supercomputers make larger models computationally tractable. Scientists can make fewer model simplifications than they otherwise could. Because supercomputers extend the detail achievable via numerical solutions to mathematical models, scientists find supercomputers useful.

Engineers Also Find Supercomputers Useful

Engineers seek practical solutions to genuine problems. They develop several design alternatives, measure the merits of each, then select the single alternative with the most merit.

Engineers develop alternatives either by testing prototypes or by mathematically modeling the alternatives. Building and testing prototypes can be expensive and time consuming. Testing variations of a prototype can require the construction of entirely new prototypes. Mathematically modeling solution alternatives can reduce or eliminate prototyping. Engineers can study variations of an alternative by changing model parameters. As an added benefit, mathematical modeling lets engineers observe events internal to the model.

Mathematically modeling engineering solutions suffers from the same shortcomings as the mathematical modeling of natural science experiments. Often, the models produce equation systems solvable only via numerical methods. Here again, the numerical methods require intractably large numbers of calculations to yield precise solutions. As with natural science research, supercomputers make larger models computationally tractable. Here also, supercomputers let engineers make fewer model simplifications, so the supercomputers extend the detail achievable via numerical solutions to the mathematical models. Thus, engineers also find supercomputers useful.

Supercomputing Topics In Current Literature

Researchers already use supercomputers in many research areas. For example, the 1990 Cornell National Supercomputing Center Abstracts of Research lists over 317 research projects. [1] The research areas include:

- astronomical sciences;
- atmospheric sciences;
- behavioral and neural sciences;
- biotic systems and resources;
- cellular biosciences;
- chemical, biochemical, and thermal engineering;
- chemistry;
- computer and computation research;
- critical engineering systems;
- earth sciences;
- electrical communications and systems engineering;
- emerging engineering technologies;
- information, robotics, and intelligent systems;
- materials research;
- mathematical sciences;
- mechanics, structures, and materials engineering;
- molecular biosciences;

- ocean sciences;
- physics;
- social and economic sciences.

As a second example, the 1990 Computer Abstracts identifies 37 articles related to vector processing, 302 related to parallel processing, and 39 related to supercomputing. [2] As a final example, the 1990 Engineering Index identifies hundreds of articles related to parallel processing, dozens related to supercomputing, and dozens related to vector processing. [3] This abundance of current literature about supercomputers, vector processing, and parallel processing implies these topics are important to modern scientists and engineers.

Supercomputing As A Driving Technology

Leaders in the field of supercomputing recognize its potential for advancing both science and engineering. Some even consider supercomputing a driving technology for increasing the economic competitiveness of the United States in the world economy. For example, Michael J. Levine, a Scientific Director of Pittsburgh Supercomputing Center writes:

> To think of supercomputers only in terms of increased speed is to miss a vital point. Their increased speed actually transforms the types of problems on which a scientist is willing to work. Decreasing the computation time from weeks to hours enables a scientist or engineer to ask much bolder questions, to try more daring and novel approaches, to include more realistic complexities, and even to replace costly and time-consuming experiments by theoretical calculation. [4]

Joanne L. Martin, Editor-in-Chief of The International Journal of Supercomputing Applications points out:

The availability of supercomputers during the last decade has been crucial to advances in the basic and applied sciences and to industrial and technological innovations. The continued development and application of supercomputers will support directly the expansion of industrial, medical, consumer, and defense segments of the U.S. and world economies. . . . This will enable the solution of computation-intensive basic research problems that previously were insurmountable, thus changing radically the patterns of many of the world's leading scientific efforts. [5]

Larry Smarr, Director of the National Center for Supercomputing Applications makes four observations about the Advanced Scientific Computing Initiative of the National Science Foundation:

First, (research) projects are broadly distributed across the Foundation's Directorate and Divisional structure. Second, it is clear that supercomputing is enabling new areas of research that were not being pursued before the initiative. Third, large blocks of time are being used on problems that only supercomputers can solve. Finally, a small amount of supercomputer time can produce a lot of good science. [6]

Erich Bloch, the former Director of the National Science Foundation, states what may be the most convincing argument about the relevance of supercomputing. He writes:

It is commonly accepted that our economy is moving rapidly into the "information age". Every year, the manufacturing sector shrinks relative to the part of the working population that generates, processes, and transmits information. The economy of the future information-oriented society will be dominated by the principal means of processing information, i.e., computers, and the principal means of transmitting information, i.e., computer networks. The supercomputer represents the leading edge of this major societal shift. This is the reason there has been so much interest recently in the development of supercomputers, both in the U.S. and abroad. The nation that maintains leadership in supercomputers will have a competitive advantage in the computer industry, and through it, an advantage in the world economy in general. This was acknowledged in a recent Fortune magazine article ("The high tech race," Oct. 16, 1986), which pointed out that almost one-half of the U.S. productivity gains in recent years have been due to technological innovation, and that a substantial part of those gains were in the computer sector. [7]

Why Study Vector And Parallel Processing

Why study vector and parallel processing? Supercomputers use vector and parallel processing to achieve their high calculation rates. Although computer vendors sell software tools that automatically vector-enable and parallel-enable target programs, these tools have limitations. The tools cannot vector- and parallel-enable programs that obscure the computational independence of the calculations by poor program coding. Also, the tools cannot infer the underlying solution algorithm and restructure the program to improve its performance. To exploit the supercomputing potential of a program fully, the programmer must understand both the program's algorithm and the principles of vector and parallel processing. The programmer then must apply these principles to improve the program's performance.

Objectives

This thesis documents a project to understand and apply the principles of vector processing and parallel processing. The project objectives are:

1. to learn about, and understand, the principles of vector and parallel processing;
2. to produce a significant reduction in CPU and elapsed time for a numerically-intensive program that produces meaningful and significant research results;
3. to document the techniques of vector processing and parallel processing in a clear, concise form suitable for reading as an introductory text;
4. to show the benefits of supercomputing to researchers and increase their interest in supercomputing;
5. to fulfill the thesis requirements for the Master of Science degree.

Supercomputing Platform Selection

I selected the IBM 3090 supercomputer as my research platform for several reasons. First, the University of Tennessee, Knoxville, has an IBM 3090 300E. Second, the U.S. Department of Commerce considers IBM 3090s (models 180 and larger) that have vector facilities (VF) to be supercomputers. [8] Third, the IBM 3090 hardware and software support both vector processing and parallel processing. Fourth, Cornell National Supercomputer Center selected the IBM 3090 as its supercomputing platform. Finally, I believe the IBM 3090 is a cost-effective platform for organizations that require supercomputing in addition to general data processing.

Chapter Contents

Chapter 1 states the relevance of vector processing and parallel processing. Chapter 2 describes the program selection criteria and the research program selection. Chapter 3 relates my vector-enabling and parallel-enabling experiences. Chapter 4 states my results, compares my objectives and results, states potential improvements, and concludes this thesis. Appendix A summarizes the supercomputing facilities available to the University of Tennessee researchers. Appendix B describes the architecture of the IBM 3090. Appendix C provides an overview of vector processing and the IBM 3090 vector facility. Appendix D describes parallel processing and the IBM 3090 parallel processing facilities. Appendix E surveys the software tools for vector-enabling and parallel-enabling programs for execution on the IBM 3090. Appendix F explains Amdahl's Law and its implications upon supercomputing. The Glossary defines important terms used throughout this thesis.

Chapter 2: Program Selection And Enabling Procedure

Project Objectives Affect Selection And Enabling

The project goals affect the program candidate selection. Given complete freedom to choose a program to work on and information about what it does, a programmer could accept or reject it based on its estimated benefits from the enabling effort. A program selected this way could get impressive reductions in central processing unit (CPU) and elapsed time. In industry, a programmer rarely has the complete freedom to choose his or her assignments. The programmer's management usually assigns a program, then expects the programmer to vector- and parallel-enable it. Although the programmer may not understand the program's underlying algorithm, the management expects him or her to reduce its CPU and elapsed time consumption.

The project goals also can affect the vector- and parallel-enabling procedure. If the programmer's only goal is to reduce the program's CPU and elapsed time consumption, he or she could substitute calls to vector- and parallel-enabled subroutines. Doing this could result in impressive reductions in CPU and elapsed time consumption, but this method suffers from two shortcomings. First, substituting calls to proprietary subroutines can reduce the program's portability. Second, the programmer would not learn about vector- and parallel-enabling.

Understanding that project goals affect both the program selection and the enabling procedure, I reviewed my project objectives. Learning about vector- and parallel-enabling is my primary objective. Reducing the CPU and elapsed time consumption of a program is my secondary objective. With these objectives in mind, I attempted to imitate what a programmer would experience in industry. I did not determine programs' functions, then accept or reject them based on my estimate of their potential benefits. When I selected the enabling procedure, I considered the program's portability.

Selection Criteria

The program selected must meet several criteria. It must consume large amounts of CPU time. The program must have compiler support for vector and parallel processing on the IBM 3090. Finally, the source code must be available.

Other characteristics can simplify its vector- and parallel-enabling and also can magnify the benefits of the enabling effort. The program should not be storage or I/O constrained. It should have computationally-independent calculations, and thus be amenable to vector- and parallel-enabling. The program should be written only with standard language constructs. It should have "hot spots." ("Hot spots" are small sections of the program that consume a disproportionately large amount of CPU time.) To increase the benefits of the enabling effort, the program should be executed repetitively. Finally, it should have an execution-length control so test runs can be short, but production runs can be long.

Selection Procedure

I sought program candidates from professors in the UTK Electrical and Computer Engineering Department. Dr. Don Bouldin suggested the SPICE circuit-simulation program. I investigated this possibility and attempted to get a copy of the SPICE program source. My investigation of the SPICE program revealed several facts. First, early versions of the SPICE program were written in FORTRAN, but the latest versions have been written in C. Because only IBM's VS FORTRAN compiler supports vector and parallel processing, the latest versions of SPICE cannot be vector- and parallel-enabled for execution on the IBM 3090. Second, the SPICE program manipulates matrices. At first glance, it appeared that SPICE might benefit from vector- and parallel-enabling. A more thorough investigation revealed that SPICE generates and manipulates sparse matrices. It stores only the non-zero elements of these matrices in a compact form. This compact storage form requires irregular data structures that are not amenable to vector processing. Finally, a student at The University of California, Berkeley, had vector-enabled SPICE for execution on a Cray supercomputer. His efforts produced only a small reduction in CPU and elapsed time. For these reasons, I eliminated SPICE as a program candidate.

Next, I examined the usage reports for the University of Tennessee's IBM 3090. I identified several project codes that consumed large amounts of CPU time and contacted their project directors. I also contacted professors, staff, and researchers whom I thought might have suitable program candidates. Some had programs amenable to vector and parallel processing; others did not.

For several reasons, I finally chose to enable programs offered by Dr. A. Jerry Baker in the UTK Engineering Science and Mechanics Department. First, Dr. Baker had several numerically-intensive program candidates and was already vector- and parallel-enabling some of them. Second, because of his interest in vector- and parallel-enabling, he would be readily available for consultation. Finally, his programs implemented finite element methods to study computational fluid dynamics. Because these topics were relevant to engineering, Dr. Baker's programs were more relevant for my research than programs unrelated to engineering.

Dr. Baker initially suggested I vector-enable a finite element modeling program developed by other engineering graduate students. I examined the source for the specific FORTRAN program he suggested and decided it was too unstructured to be vector-enabled manageably. I approached Dr. Baker for a different finite element modeling program; this time he suggested the AKCESS program developed by the Computational Mechanics Corporation. Dr. Baker described its capabilities and stated that professional programmers had developed AKCESS. Impressed with its capabilities and convinced it was well-structured and suitable for vector-enabling, I decided to vector-enable the AKCESS program.

Enabling Procedure

The vector- and parallel-enabling procedure has several steps. [9] [10] These are:

- port the program to the target system;
- attempt automatic vector- and parallel-enabling;
- identify execution hot spots;
- vector- and parallel-enable the hot spots;
- tune the program.

Before beginning the vector- and parallel-enabling, the programmer should execute the program on the source system to get control answers. After each step, the programmer should execute the program and compare the answers with the control answers. Comparable answers verify the enabling steps have not introduced errors. (Naturally, for the control answers to be useful, the program must correctly execute on the source system.) Finally, after each step, the programmer should collect the CPU, vector facility, and elapsed time information.

To port the program to another platform, the programmer must move, compile, and link the source to create an executable module for the target system. The programmer can easily do this if the program does not contain source system-specific language extensions. If the program contains source system-specific language extensions, the programmer must remove them. The programmer also must remove source system-specific file names and data access methods.

The programmer should attempt automatic vector- and parallel-enabling. This may provide satisfactory reductions in CPU and elapsed time consumption with almost no effort. To do this, the programmer simply must select the proper compiler options and compile the program.

If the automatic enabling does not provide reductions in CPU and elapsed time consumption that the programmer is satisfied with, he or she should identify the program's execution hot spots. Execution monitors and timers help find the hot spots. To maximize the benefit/cost ratio of the enabling efforts, the programmer should concentrate his or her efforts on the hottest spots.

The programmer may vector- and parallel-enable the hot spots either by substituting calls to vector- and parallel-enabled subroutines or by manual intervention. Vendor subroutines probably have been tuned to achieve the best performance on a given machine, so substituting calls can be quick and easy and still yield impressive results. This can maximize the benefit/cost ratio of the enabling effort. The programmer would substitute calls to vendor subroutines only into programs that will not be ported to other platforms. For portable programs, the programmer must manually vector- and parallel-enable the hot spots. Compiler messages and data dependence analyzers help the programmer identify inhibitors to vector and parallel processing. The programmer may have to rewrite program sections, or possibly the entire program, to implement algorithms more amenable to vector and parallel processing.

Finally, the programmer may tune the program to maximize its performance on the target platform. Program tuning promotes better use of the target machine's architecture and resources.

Chapter 3: Vector- And Parallel-Enabling Experiences

Programs Studied

For my thesis research, I studied two programs: AKCESS and a benchmark program. AKCESS is a "real world" problem; it was written to solve a problem, not specifically to vector- and parallel-enable. The benchmark program helped estimate the calculation rates of the IBM 3090 central processing unit (CPU) and vector facility (VF).

AKCESS is a set of programs that form a general-purpose finite element modeling tool. It is a proprietary program owned by the Computational Mechanics Corporation. Unlike other finite element modeling programs, AKCESS can model almost any differential equation. As part of my thesis research, I studied the single program that implements its computational engine.

I developed the benchmark program to exploit the IBM 3090 VF. After tuning the program to achieve the best-case performance on the IBM 3090 VF, I executed it in scalar mode. From the CPU/VF timing information yielded by these runs, I estimated the computational rates of the IBM 3090 VF and CPU. With this information, I estimated the vector/scalar speed ratio of the IBM 3090.

AKCESS Experiences

Development Environment

I modified AKCESS and executed scalar test runs on an IBM 3081-D32 running the VM/HPO Version 1, Release 5, Modification level 0 (V1R5M0) operating system. On this system, I compiled AKCESS with the VS FORTRAN V2R4M0 compiler. I worked primarily in this hardware/software environment because of my familiarity with it.

I executed the vector runs on an IBM 3090-300E, running the MVS/SP V3R1M3 operating system and the TSO/E V2R1M1 timesharing monitor. In this environment, I compiled AKCESS with the VS FORTRAN V2R4M0 compiler.

Preliminary Work

First, I ported ACCESS to the IBM systems. I moved AKCESS from a Silicon Graphics Iris 3130 workstation at the Computational Mechanics Corporation to the VM/HPO system via the Internet File Transfer Protocol (FTP).

After moving the program to the VM/HPO system, I edited the program and data files to remove imbedded file names. I changed the file names to data-definition names. I also determined the dataset record format and record length characteristics of the files.

After removing the imbedded file names, I compiled the program and executed it on the VM/HPO system. I compared the results with the results from the original system to verify the program's correct operation on the VM/HPO system.

After verifying the program's correct operation on the VM/HPO system, I moved it to the MVS/SP system. Again I compiled and executed the program, then compared its results with the original results.

After verifying the program's correct operation on the MVS/SP system, I found its execution "hot spots"; i.e., those subroutines and statements that consume a disproportionately large share of the CPU time. To do this, I first executed the program with the VS FORTRAN Interactive Debugger (IAD) enabled to find the hot subroutines. I then executed the program with the VS FORTRAN Interactive Debugger (IAD) enabled to find the hot statements within the hot subroutines. This procedure let me quickly locate the hot spots and print the analysis only for the hottest subroutines.

For my test case, two subroutines consumed about 81% of the total CPU time. The SLINEG subroutine consumed about 61% of the total. The SLINEJ subroutine consumed about 20% of the total. After finding the hot subroutines, I executed the program with the IAD enabled to find the hot statements in the SLINEG subroutine. Several loops of 20 statements or less each consumed about 3-5% of the entire CPU time.

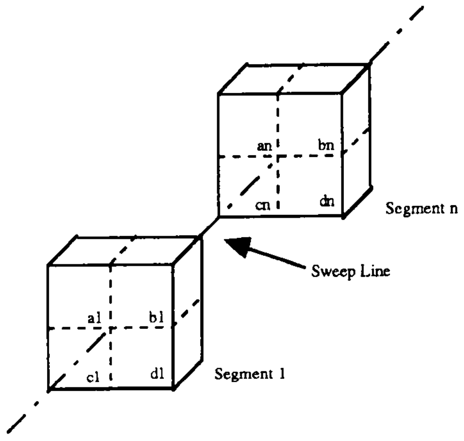
After locating the execution hot spots, I attempted automatic vector-enabling. The compiler could vector-enable only a small fraction of the SLINEG subroutine automatically. Several factors inhibited the automatic vector-enabling. Some loops contained CALL statements. Some loops contained IF...THEN...GOTO statements. Some loops contained READ, WRITE, or other I/O statements. Some loops contained data recurrences. Some loops contained too few iterations to benefit from vector execution. This absence of a significant amount of automatically vector-enablable code implied the SLINEG algorithm required restructuring to exploit vector processing.

Overview Of SLINEG Operation

Finite element algorthms set up a system of equations $Ax = b$. In this equation, A is a matrix of known coefficients, b is a vector of known coefficients, and x is a vector of unknown values. The algorithms solve for x by inverting A and multiplying b by the inverse. In practice, finite element modeling programs use other solution methods because the matrix A is a large sparse matrix. Storing A would require large amounts of machine storage, most of which would contain 0 elements. Also, inverting the matrix A would require an intractably large number of calculations.

AKCESS reduces the storage and number of calculations required to model a differential equation by building partial solutions and merging them. AKCESS builds the partial solutions along series of nodes in the finite element mesh; these are called "sweep lines." For each finite element that borders a sweep line, the construction of the partial solution requires many calculations. These must be repeated for all finite elements that border the sweep line.

The scalar SLINEG subroutine sequences through all finite elements that border a sweep line. It performs all calculations for a single finite element before proceeding to the next finite element. Also, it processes finite elements with boundary sets as it encounters the finite element. The vector-enabled SLINEG subroutine identifies all finite elements that border the sweep line and treats this list as a vector. It performs a single calculation for all of the finite elements before proceeding to the next calculation. Finally, it processes boundary sets. Figure 1 shows the differences between the scalar and vector-enabled SLINEG subroutines.



Original SLINEG

```
    DO 30, SEGMENT = 1,N
       DO 20, ELEMENT = A,D
          DO 10, CALCULATIONS = 1,M
                +----------------------------------+
                | PERFORM AN                       |
                | INDIVIDUAL CALCULATION           |
                +----------------------------------+
                +----------------------------------+
                | CALCULATE BOUNDARY               |
                | CONDITIONS AS WE                 |
                | ENCOUNTER THEM                   |
                +----------------------------------+
10              CONTINUE
             +-------------------------------------+
             | MERGE RESULTS FROM INDIVIDUAL       |
             | CALCULATIONS INTO A SINGLE RESULT   |
             | FOR THE FINITE ELEMENT              |
             +-------------------------------------+
             +-------------------------------------+
             | DEPOSIT ELEMENTAL RESULTS INTO      |
             | NODES THAT BOUND THE SEGMENT        |
             +-------------------------------------+
20           CONTINUE
30        CONTINUE
```

Restructured SLINEG

```
    +--------------------------------------------+
    | IDENTIFY ALL FINITE ELEMENTS               |
    | THAT BORDER THIS SWEEP LINE                |
    +--------------------------------------------+
    +--------------------------------------------+
    | MAKE A VECTOR FROM THESE FINITE            |
    | ELEMENTS: A1-AN, B1-BN, C1-CN, D1-DN       |
    +--------------------------------------------+
    DO 10, CALCULATIONS = 1,M
       +-----------------------------------------------+
       | PERFORM AN INDIVIDUAL CALCULATION             |
       | FOR ALL FINITE ELEMENTS A1-DN IN THE          |
       | VECTOR AND SAVE THE RESULTS                   |
       | THIS IS A VECTOR LOOP                         |
       +-----------------------------------------------+
10     CONTINUE
    +--------------------------------------------+
    | FOR ALL FINITE ELEMENTS IN THE VECTOR      |
    | A1-DN, MERGE THE RESULTS FROM THE          |
    | INDIVIDUAL CALCULATIONS INTO A SINGLE      |
    | RESULT FOR EACH FINITE ELEMENT             |
    | THIS IS A VECTOR LOOP                      |
    +--------------------------------------------+
    +--------------------------------------------+
    | DEPOSIT ELEMENTAL RESULTS                  |
    | INTO NODES ON THE SWEEP LINE               |
    | THIS IS A VECTOR LOOP                      |
    +--------------------------------------------+

    +--------------------------------------------+
    | IDENTIFY THE FINITE ELEMENTS WHERE         |
    | BOUNDARY CONDITIONS APPLY                  |
    | THIS IS A SCALAR LOOP                      |
    +--------------------------------------------+
    +--------------------------------------------+
    | CALCULATE THE BOUNDARY CONDITIONS          |
    | THIS IS A SCALAR LOOP                      |
    +--------------------------------------------+
    +--------------------------------------------+
    | DEPOSIT THE BOUNDARY CONDITIONS INTO       |
    | THE NODES THAT BOUND THE SWEEP LINE        |
    | THIS IS A SCALAR LOOP                      |
    +--------------------------------------------+
```

Figure 1. Differences Between Scalar And Vector SLINEG Algorithms

Vector-Enabling SLINEG

Vector-enabling the SLINEG subroutine required restructuring its algorithm. To vector-enable the SLINEG subroutine, I divided the AKCESS program into two pieces: the SLINEG subroutine and "everything else." I compiled "everything else" once and saved its object module. I modified and compiled the SLINEG subroutine as often as needed. To execute and test the AKCESS program, I put the two object modules together with the link-editor.

As I vector-enabled the SLINEG subroutine, I put the vector-enabled code in the same FORTRAN source file with the original code. I put the vector-enabled code before the original code and insured it did not change variables required by the original. This arrangement let me execute both the vector-enabled and original code together as part of the same program. I inserted WRITE statements where appropriate to get debugging output. With this output, I could compare the results from the vector-enabled code against results from the original code to verify the correct operation of the vector-enabled code.

Refining And Tuning The Vector-Enabled SLINEG Subroutine

As the work progressed and I understood the principles of vector- and parallel-enabling better, I refined and tuned the vector-enabled SLINEG code. The refinements included adding a conditional branch around the vector-enabled code, removing work from within loops, and combining vector-enabled DO loops.

I added a conditional branch around the vector-enabled SLINEG code. If the sweep line length is too short to benefit from vector execution, SLINEG executes the scalar code. This optimizes the performance of the SLINEG subroutine regardless of the sweep line length.

I removed program statements from within a loop. These statements compute the same results each iteration. I moved the statements outside the loop and stored the computed results. The program computes these results once, then repeatedly uses them, and thus saves CPU and VF time.

In the original vector-enabled subroutine, where several DO loops with identical index bounds followed each other, I combined the executable statements into a single DO loop. In these cases, earlier DO loops computed results and stored the results in an array. Later DO loops read this array as inputs to their calculations. Because the loop indices had the same bounds, I could combine the loops. I removed data recurrences by introducing temporary scalar variables. When compiled, these temporary scalar variables expand into a vector of temporary variables. This refinement saves CPU and VF time because fewer DO loops are initialized. Also, the program retains intermediate results in vector registers instead of accessing main storage. Figure 2 shows a set of DO loops from the original vector-enabled SLINEG routine. Figure 3 shows an optimized version of these loops.

I considered converting an outer DO loop in the vector-enabled code into a parallel loop. This would introduce parallelism into the SLINEG subroutine. For the DO loop I considered, this may not be the best way to introduce parallelism into AKCESS. The amount of computation executed within the parallel loop would be small relative to the parallel loop overhead. Thus, the loop overhead might be a large fraction of the total CPU time consumed by the parallel loop.

```
LEVEL 2.4.0 (AUG  1989)        VS FORTRAN            JUN 23, 1991  02:41:29  NAME:OLDCS

                                           REPORT(XLIST)  VECTORIZATION ANALYSIS

 ISN FLAG NESTING *....*...1.........2.........3.........4.........5.........6.........7.*......

0001                      PROGRAM    OLDCS                                               OLD000
0002                      INTEGER*4  TMPINT(800,15), ELMINT(800,15)                      OLD000
0003                      REAL*4     TMPREL(800,15), RZ(800000), CS(10000)               OLD000
                 C*********************************************************************OLD000
                 C**                                                                 **OLD000
                 C**  FUNCTION:  GIVEN A TERM NUMBER AND A LIST OF ELEMENTS, THIS    **OLD000
                 C**             ROUTINE EXTRACTS THE MATERIAL COEFFICIENT C'S.      **OLD000
                 C**                                                                 **OLD000
                 C**  OUTPUTS:  TMPREL(I,1) - CONTAINS THE MATERIAL COEFFICIENTS.    **OLD000
                 C**                                                                 **OLD001
                 C*********************************************************************OLD001
                                                                                       OLD001
                 C*********************************************************************OLD001
                 C**                                                                 **OLD001
                 C**  INITIALIZE THE TEMPORARARY ARRAY OF MATERIAL NUMBERS.          **OLD001
                 C**                                                                 **OLD001
                 C*********************************************************************OLD001
0004 VECT +-------        DO 1007, I=1, ELCNT                                            OLD001
0005      |_______            TMPINT(I,2) = 1                                            OLD001
                 C*********************************************************************OLD002
                 C**                                                                 **OLD002
                 C**  IF NECESSARY, GET THE MATERIAL NUMBERS FOR EACH ELEMENT.       **OLD002
                 C**                                                                 **OLD002
                 C*********************************************************************OLD002
0007                      IF (IMTRL .GT. 0) THEN                                         OLD002
                 C****GET POINTERS INTO RZ FOR THE MATERIAL NUMBERS*********************OLD002
0008 VECT +-------            DO 1017, I=1, ELCNT                                        OLD002
0009      |_______                TMPINT(I,1) = IMTRL + ELMINT(I,2)                      OLD002
                 C****GET THE MATERIAL NUMBERS******************************************OLD003
0011 VECT +-------            DO 1027, I=1, ELCNT                                        OLD003
0012      |_______                TMPINT(I,2) = RZ(TMPINT(I,1))                          OLD003
0014                          END IF                                                     OLD003
                 C*********************************************************************OLD003
                 C**                                                                 **OLD003
                 C**  GET SUBSCRIPTS THAT POINT INTO THE CS ARRAY.                   **OLD003
                 C**                                                                 **OLD003
                 C*********************************************************************OLD004
0015 VECT +-------        DO 1037, I=1, ELCNT                                            OLD004
0016      |_______            TMPINT(I,3) = (TMPINT(I,2) - 1) * KMTRXF + TERMNO          OLD004
                 C*********************************************************************OLD004
                 C**                                                                 **OLD004
                 C**  GET THE MATERIAL COEFFICIENTS.                                 **OLD004
                 C**                                                                 **OLD004
                 C*********************************************************************OLD004
0018 VECT +-------        DO 1047, I=1, ELCNT                                            OLD004
0019      |_______            TMPREL(I,1) = CS(TMPINT(I,3))                              OLD005
                 C*********************************************************************OLD005
                 C**                                                                 **OLD005
                 C**  END OF THE CONSTRUCTION OF THE MATERIAL COEFFICIENT LIST.      **OLD005
                 C**                                                                 **OLD005
                 C*********************************************************************OLD005
0021                      STOP                                                           OLD005
0022                      END                                                            OLD005
```

Figure 2. Original Vector-Enabled Loop

```
LEVEL 2.4.0 (AUG  1989)        VS FORTRAN           JUN 23, 1991  02:41:24  NAME:NEWCS

                                            REPORT(XLIST)  VECTORIZATION ANALYSIS

ISN FLAG NESTING *....*...1.........2.........3.........4.........5.........6.........7.*......

0001                       PROGRAM    NEWCS                                              NEW000
0002                       INTEGER*4  ELMINT(800,15)                                     NEW000
0003                       REAL*4     TMPREL(800,15), RZ(800000), CS(10000)              NEW000
                  C*******************************************************************NEW000
                  C**                                                               **NEW000
                  C**  FUNCTION:  GIVEN A TERM NUMBER AND A LIST OF ELEMENTS, THIS  **NEW000
                  C**             ROUTINE EXTRACTS THE MATERIAL COEFFICIENT C'S.    **NEW000
                  C**                                                               **NEW000
                  C**  OUTPUTS:  TMPREL(I,1) - CONTAINS THE MATERIAL COEFFICIENTS.  **NEW000
                  C**                                                               **NEW001
                  C*******************************************************************NEW001
                                                                                     NEW001
                  C*******************************************************************NEW001
                  C**                                                               **NEW001
                  C**  GET THE MATERIAL COEFFICIENTS.                               **NEW001
                  C**                                                               **NEW001
                  C*******************************************************************NEW001
0004 VECT +-------        DO 1007, I=1, ELCNT                                            NEW001
0005      |                  ITM100 = IMTRL + ELMINT(I,2)                                NEW001
0006      |                  ITM101 = RZ(ITM100)                                         NEW002
0007      |                  IF (IMTRL .LE. 0)  ITM101 = 1                               NEW002
0009      |                  ITM102 = (ITM101 - 1) * KMTRXF + TERMNO                     NEW002
0010      |_______           TMPREL(I,1) = CS(ITM102)                                    NEW002
                  C*******************************************************************NEW002
                  C**                                                               **NEW002
                  C**  END OF THE CONSTRUCTION OF THE MATERIAL COEFFICIENT LIST.    **NEW002
                  C**                                                               **NEW002
                  C*******************************************************************NEW002
0012                         STOP                                                        NEW003
0013                         END                                                         NEW003
```

Figure 3. Optimized Vector-Enabled Loop

SLINEG Execution

By the research completion deadline, I had not vector-enabled the SLINEG subroutine completely. The part of SLINEG I had vector-enabled performed about 80% of the computation of the original. Although the vector-enabling was incomplete, I was able to estimate the speedup due to the vector-enabling.

To estimate the speedup, I first executed AKCESS with the scalar-only SLINEG subroutine. I executed it once without the VS FORTRAN Interactive Debugger (IAD) enabled and once with the IAD enabled to find the hot subroutines. These runs yielded the total CPU time required to execute my test finite element model. They also yielded the CPU time percentage consumed by the SLINEG subroutine and the

CPU time percentage consumed by the rest of the AKCESS program. With this information, I calculated the CPU time consumed both by the SLINEG subroutine and by the rest of the AKCESS program.

Next, I executed AKCESS with the version of SLINEG that contained both the vector-enabled code and the scalar code. This execution yielded a CPU time number that included the additional CPU time consumed by the vector-enabled code. I subtracted the CPU times from the scalar-only SLINEG runs from these CPU times. This subtraction yielded the CPU time consumed by the vector-enabled portion of the SLINEG subroutine. I then calculated the speedup of the two versions of SLINEG. Because the vector-enabling was incomplete, the calculated speedup was only an estimate.

Benchmark Program Experiences

Development Environment

I developed the program on an IBM 3090-300E running the MVS/SP V3R1M3 operating system and the TSO/E V2R1M1 timesharing monitor. I compiled the program with the VS FORTRAN V2R4M0 compiler and the Parallel FORTRAN Prototype compiler V1R1M3.

Program Development

I wrote the benchmark program to yield the greatest calculation rate on the IBM 3090 VF. I ran it in both vector and scalar mode. The two runs let me estimate the vector calculation rate, the scalar calculation rate, and the vector/scalar speed ratio of the IBM 3090.

From the onset of the program development, I wanted to get IBM 3090 VF multiply and add instructions (VMAxx opcode mnemonics) into the program. These instructions achieve the fastest calculation rate on the IBM 3090 VF because they execute two floating-point calculations per machine cycle.

I wrote and revised the benchmark program many times, but the calculation rate of the early versions did not approach the theoretical peak calculation rate of the IBM 3090 VF. To determine if the compiler generated VMAxx instructions in the object module, I instructed it to produce assembly-language listings of the compiled FORTRAN programs. The listings revealed the compiler was not generating the VMAxx instructions in the object module.

I researched the techniques for writing DO loops for which the compiler generated the VMAxx instructions. [11] [12] With this information, I wrote, compiled, and ran a program that contained the VMAxx instructions. The new program's calculation rate still did not approach the theoretical peak calculation rate of the IBM 3090 VF; main storage accessing became the calculation rate-limiting factor. After I reduced the vector length to a length that stayed CPU cache-resident, the calculation rate increased. Finally I set the vector length equal to the vector section size of the IBM 3090-300E; again the calculation rate increased. Setting the vector length equal to the vector section size amortizes the VF pipeline initialization penalties over the longest single vector section, so it minimizes the per-calculation time.

For several reasons, I stopped attempting to improve the program's vector performance. First, the compiler was generating small, fast-executing loops that contained the VMAxx instructions. Second, I could not increase the calculation rate by reducing the number of main storage accesses. Finally, recoding the program in System/370 assembly language would be required to gain additional performance.

After achieving the greatest possible single-processor vector execution rate, I modified the benchmark for parallel execution. I added a parallel DO loop and changed array dimensions to let different CPUs process different parts of the arrays. I recompiled the program with the Parallel FORTRAN Prototype compiler and executed the test cases.

Benchmark Execution

I executed 12 different test cases of the benchmark program. I executed each test case twice so I could average the CPU/VF time values. When I executed the test cases, all programs other than the operating system services were stopped. Also, I executed only one test case at a time. These two precautions let each parallel test case consume all the CPU/VF time they required.

The individual test cases differed by whether they used single-precision or double-precision operands, whether they used scalar or vector machine instructions, and whether they concurrently executed on one, two, or three CPU/VFs. Specifically, the test cases were:

- Single-precision scalar, one CPU/VF.
- Single-precision scalar, two CPUs/VFs.
- Single-precision scalar, three CPUs/VFs.
- Double-precision scalar, one CPU/VF.
- Double-precision scalar, two CPUs/VFs.
- Double-precision scalar, three CPUs/VFs.
- Single-precision vector, one CPU/VF.
- Single-precision vector, two CPUs/VFs.
- Single-precision vector, three CPUs/VFs.
- Double-precision vector, one CPU/VF.
- Double-precision vector, two CPUs/VFs.
- Double-precision vector, three CPUs/VFs.

Chapter 4: Results And Conclusions

AKCESS Results

Table 1 shows the raw central processing unit (CPU) and vector facility (VF) timing information from the AKCESS runs. In this table, the "SLINEG Version" column specifies whether the AKCESS run used the scalar-only or vector-enabled version of the SLINEG subroutine. The "Run Number" column specifies the trial number of the AKCESS execution. The MVS/SP operating system reported these CPU and VF times after the program completed execution. The CPU times include the VF times.

Table 1. Raw Timing Information For AKCESS

SLINEG version	Run Number	CPU time (ss.ff)	VF time (ss.ff)
Scalar-Only	1	15.80	0.15
	2	15.75	0.15
	3	15.70	0.15
	4	15.70	0.15
	Average	15.74	0.15
Vector-Enabled	1	20.25	2.24
	2	20.22	2.24
	3	20.18	2.23
	4	20.12	2.24
	Average	20.19	2.24

For an execution of the scalar-only AKCESS program, the VS FORTRAN Interactive Debugger (IAD) reported that the scalar-only SLINEG subroutine consumed 60.72% of the total CPU time. It also reported the rest of the AKCESS program consumed 39.28% of the total CPU time. An execution of the scalar-only AKCESS program consumed an average of 15.74 seconds of CPU time, of which 0.15 seconds were VF time. Thus, the CPU times for the scalar-only SLINEG subroutine and the rest of the AKCESS program were:

```
SLINEG subroutine:   (60.72 / 100.0) x 15.74 = 9.56 secs.

Rest of AKCESS:      (39.28 / 100.0) x 15.74 = 6.18 secs.
```

Because I had not vector-enabled the SLINEG subroutine completely, I ran AKCESS again with the SLINEG subroutine that contained *both* the scalar-only and vector-enabled code. Due to the added vector-enabled code, the CPU time consumption *increased* over the scalar-only AKCESS runs. These executions consumed an average of 20.19 seconds of CPU time, of which 2.24 seconds were VF time. I calculated the CPU and VF time of the vector-enabled SLINEG code by subtracting:

```
VF time:    ( 2.24 -  0.15) = 2.09 secs.

CPU time:   (20.19 - 15.74) = 4.45 secs.
```

I divided these numbers to calculate the vector-execution fraction of the vector-enabled SLINEG code:

```
(2.09 VF secs. / 4.45 CPU secs.) x 100 = 47%  vector execution
```

If I had completed the vector-enabling of the SLINEG subroutine, I could have deleted its scalar-only code. This would have removed the SLINEG scalar-execution CPU time component from the AKCESS runs. Due to its vector-enabling, the SLINEG speedup would have been:

```
9.56 secs. / 4.45 secs.  =  2.15
```

The CPU time consumption of the vector-enabled AKCESS program would have been:

```
 15.74 secs.  (CPU time of the scalar-only AKCESS program)
- 9.56 secs.  (CPU time of the scalar-only SLINEG code)
+ 4.45 secs.  (CPU time of the vector-enabled SLINEG code)
-----------------------------------------------------------
 10.63 secs.
```

Thus, the overall speedup of the AKCESS program would be:

```
(15.74 scalar secs. / 10.63 vector secs.)  =  1.48
```

This estimated overall speedup of 1.48 is artificially low. The model mesh was aligned with the x, y, and z axes. This alignment caused many calculations to have results of value zero. Although the scalar-only SLINEG code skipped calculations when it detected zero results, the vector-enabled SLINEG code did not. During execution, the vector-enabled SLINEG code calculated 3 times as many results as the scalar-only SLINEG code. Two-thirds of these results were zero.

If the model mesh had been skewed from the x, y, and z axes, the number of calculations with non-zero results would have increased, potentially by a factor of 3. This would have increased the CPU time consumption of the scalar-only SLINEG code. The CPU time consumption of the vector-enabled SLINEG code would have remained constant. For the skewed-mesh model, the CPU time consumption for the scalar-only SLINEG code potentially would have been:

```
9.56 secs. x 3 = 28.68 secs.
```

The SLINEG speedup due to the vector-enabling would have been:

```
28.68 secs. / 4.45 secs. = 6.44
```

The overall CPU time requirement for the scalar-only AKCESS would have been:

```
 28.68 secs.  (CPU time of the scalar-only SLINEG code)
+ 6.18 secs.  (CPU time for the rest of AKCESS)
-------------------------------------------------------
 34.86 secs.
```

Finally, the overall program speedup would have been:

```
(34.86 scalar secs. / 10.63 vector secs.) = 3.28
```

Benchmark Results

Table 2 shows the raw CPU and VF timing information for the scalar benchmark runs. Table 3 shows the raw CPU and VF timing information for the vector benchmark runs. In these tables, the "Precision" column specifies the numerical precision of the operands in the benchmark. The "n" column specifies the number of CPUs the benchmark executed on. The "Run Number" column specifies the trial number of the benchmark execution. The "%" column specifies the percentage of the entire CPU time available on the three-processor IBM 3090-300E the benchmark consumed. The MVS/SP operating system reported the CPU, VF, and elapsed times after the program completed execution. The CPU times include the VF times.

Table 4 shows the average calculation rates of the benchmark programs. In this table, the "Benchmark" column specifies the numerical precision of the operands in the benchmark and whether the benchmark was a scalar or vector benchmark. The "n" column specifies the number of CPUs the benchmark executed on. The "Average Time" column specifies the average time value for the benchmark. I copied this information from the italicized values in the benchmark raw timing tables. The "Average MFLOPS" column specifies the millions of floating-point operations per second the benchmark executed.

In Table 4, for the single-CPU tests, I computed the average calculation rate by dividing the number of calculations (7680 million) by the average CPU time. For the multiple-CPU tests, I computed the average calculation rate by dividing the number of calculations (7680 million) by the average elapsed time. Dividing the number of calculations by the elapsed time is a valid method for computing the calculation rate. This is true because the multiple CPUs, working together, deliver this calculation rate to the benchmark program.

Table 5 shows the vector/scalar speed ratios seen by the benchmark programs. In this table, the "Precision" column specifies the numerical precision of the operands in the benchmark. The "Vector MFLOPS" and "Scalar MFLOPS" columns specify the average vector and scalar calculation rates. The "Ratio" column specifies the vector/scalar speed ratio.

Table 2. Raw Timing Information For Scalar Benchmarks

Precision	n	Run Number	%	CPU time (mm:ss.ff)	VF time (mm:ss.ff)	Elapsed time (mm:ss.ff)
Single	1	1	32	12:01.48	not applicable	12:32.00
		2	32	12:01.52	not applicable	12:33.00
		Average	32	***12:01.50***	not applicable	12:32.50
	2	1	63	12:01.69	not applicable	06:20.00
		2	63	12:01.75	not applicable	06:21.00
		Average	63	12:01.72	not applicable	***06:20.50***
	3	1	92	12:02.15	not applicable	04:21.00
		2	92	12:02.16	not applicable	04:22.00
		Average	92	12:02.16	not applicable	***04:21.50***
Double	1	1	32	11:54.72	not applicable	12:26.00
		2	32	11:54.67	not applicable	12:25.00
		Average	32	***11:54.70***	not applicable	12:25.50
	2	1	63	11:55.04	not applicable	06:17.00
		2	63	11:55.00	not applicable	06:17.00
		Average	63	11:55.02	not applicable	***06:17.00***
	3	1	92	11:55.60	not applicable	04:19.00
		2	92	11:55.55	not applicable	04:20.00
		Average	92	11:55.58	not applicable	***04:19.50***

Table 3. Raw Timing Information For Vector Benchmarks

Precision	n	Run Number	%	CPU time (mm:ss.ff)	VF time (mm:ss.ff)	Elapsed time (mm:ss.ff)
Single	1	1	32	03:41.27	03:11.58	03:53.00
		2	32	03:41.23	03:11.58	03:51.00
		Average	32	***03:41.25***	03:11.58	03:52.00
	2	1	63	03:35.85	02:59.02	01:54.00
		2	63	03:36.06	02:59.09	01:54.00
		Average	63	03:35.96	02:59.06	***01:54.00***
	3	1	92	02:27.92	02:00.92	00:54.00
		2	92	02:30.34	02:02.73	00:54.00
		Average	92	02:29.13	02:01.83	***00:54.00***
Double	1	1	32	02:34.74	02:12.02	02:42.00
		2	32	02:34.70	02:12.01	02:41.00
		Average	32	***02:34.72***	02:12.02	02:41.50
	2	1	62	02:17.24	01:51.01	01:13.00
		2	62	02:31.12	02:02.08	01:20.00
		Average	62	02:24.18	01:56.55	***01:16.50***
	3	1	91	01:43.81	01:19.17	00:38.00
		2	91	01:26.97	01:07.34	00:31.00
		Average	91	01:35.39	01:13.26	***00:34.50***

Table 4. Benchmark Calculation Rates

Benchmark	n	Average Time (mm:ss.ff)	Average MFLOPS
Single-precision scalar	1	12:01.50	10.64
	2	06:20.50	20.18
	3	04:21.50	29.37
Double-precision scalar	1	11:54.70	10.75
	2	06:17.00	20.37
	3	04:19.50	29.60
Single-precision vector	1	03:41.25	34.71
	2	01:54.00	67.37
	3	00:54.00	142.22
Double-precision vector	1	02:34.72	49.64
	2	01:16.50	100.39
	3	00:34.50	222.61

Table 5. Vector/Scalar Speed Ratios

Precision	n	Vector MFLOPS	Scalar MFLOPS	Ratio
Single	1	34.71	10.64	3.26
	2	67.37	20.18	3.34
	3	142.22	29.37	4.84
Double	1	49.64	10.75	4.62
	2	100.39	20.37	4.93
	3	222.61	29.60	7.52

Payoff From Vector-Enabling AKCESS

Several benefits result from the vector-enabling. First, because the overall program speedup could be as high as 3.28, a finite element analyst potentially may model with 3.28 times as many nodes. For example, if the CPU time requirements had constrained a model to 1,000,000 or fewer nodes, CPU time requirements now may constrain it to 3,280,000 nodes. Second, x, y, and z axes alignment no longer constrains the models. A non-axes-aligned model requires no more CPU time to solve than an axes-aligned model. The removal of this constraint lets a finite-element analyst focus upon the model instead of yet another aspect of its computational tractability. Third, existing finite element models require less CPU time to solve. Thus, they cost less to solve.

I cannot easily and accurately determine the financial break-even for the vector-enabling of the SLINEG subroutine. The vector-enabling cost is fixed; it includes two man-months of programmer time (free, in this case), and about $4100 in machine charges. The machine charge savings resulting from the vector-enabling vary due to many factors. These factors include the finite element model size, the number of program runs, the machine cost per CPU hour, and other factors.

Finally, the vector-enabling adds value to the AKCESS program product. This potentially increases its sale price.

Deliverables

My research has produced the following deliverables:

1. A code review of the scalar SLINEG subroutine. This code review found programming errors in the scalar SLINEG subroutine. It also identified ways to improve the performance of the scalar SLINEG subroutine.
2. A mostly-complete vector-enabled SLINEG subroutine. This vector-enabled code can be completed, enhanced, and ported to the SLINEJ subroutine.
3. A program suitable for benchmarking other IBM System/370 or System/390 processors.
4. Estimates of the scalar and vector calculation rates of the IBM 3090.

Closing The Loop

This section "closes the feedback loop." It compares the actual project results with the stated project objectives.

Objective 1: To learn about, and understand, the principles of vector and parallel processing. I achieved this objective. I now understand the principles of vector and parallel processing. Also, I can identify and remove inhibitors to vector- and parallel-enabling.

Objective 2: To produce a significant reduction in CPU and elapsed time for a numerically-intensive program that produces meaningful and significant research results. Although I showed a significant reduction in CPU time for a numerically-intensive program, my code is not ready for production use.

Objective 3: To document the techniques of vector processing and parallel processing in a clear, concise form suitable for reading as an introductory text. I achieved this objective. This thesis is the finished product.

Objective 4: To show the benefits of supercomputing to researchers and increase their interest in supercomputing. Although my results showed the benefits of supercomputing, I did not increase the faculty interest in supercomputing.

Programmer Education

During my research, I learned many details of vector and processing. I learned how vector and parallel processing function, vector- and parallel-enabling procedures, and how to identify and remove inhibitors to vector- and parallel-enabling. I know the vector- and parallel-enabling software tools for the IBM 3090 and am familiar with their use. I now can develop computationally-independent algorithms easily and translate these algorithms into vector- and parallel-enabled programs. Learning how to "think for computational independence" may be my single greatest benefit from the research. I learned several criteria to look for in an algorithm when developing a vector- and parallel-enabled program. First, could the algorithm be vector- and parallel-enabled? Second, are the benefits worth the cost?

Could The Algorithm Be Vector- And Parallel-Enabled

To decide if the algorithm could be vector- and parallel-enabled, I would need additional information. Could the program be written in FORTRAN? Would the program use arrays, even small arrays (more than 20 elements in the largest dimension?) Does the algorithm contain computational independence? Specifically, given any array subscript, could I calculate the results associated with that subscript without referencing results associated with any other subscript?

Recommendations: If all these questions have affirmative answers, then examine the enabling costs and benefits. If any of these questions has a negative answer, then do not consider vector- and parallel-enabling further.

If the program could be written in FORTRAN, then it could be vector- and parallel-enabled for execution on the IBM 3090. If the program would use arrays, then it would satisfy the data requirements for the enabling. It would contain identical data elements accessible via indexing. The program must have arrays larger than 20 elements in the largest dimension to benefit economically from vector execution on the IBM 3090. Affirmative answers to the last two questions suggest the program contains the computational independence required for enabled execution.

Are The Benefits Worth The Cost

To decide if the enabling benefits justify the enabling cost, I also would need additional information. How much CPU time would the program consume over its lifetime? Would the program consume much CPU time manipulating the arrays? How vector- and parallel-literate is the programmer? Is programmer education one enabling benefit? How important is pushing the limits of the computational science in the researcher's field?

Recommendations: If the programmer is vector- and parallel-literate, then enable the program. The enabling will cost very little. If the programmer is vector- and parallel-illiterate, but programmer education is one benefit, then enable the program. The enabling may be costly, but the programmer will become vector- and parallel-literate. The programmer's experience can be reused. If the programmer is vector- and parallel-illiterate, and programmer education is not a benefit, then only enable programs that consume large amounts (i.e., hundreds of hours) of CPU time. Here, the machine charge savings resulting from the enabling must be greater than the cost of the enabling. If pushing the limit of the computational science in your field is important, then enable the program. You may be unable to compute the result any other way.

The first two questions help assess the benefits of the enabling effort. Programs that consume little CPU time over their lifetime must cost very little to vector- and parallel-enable. Programs that consume large amounts of CPU time may cost less to vector- and parallel-enable than to execute non-enabled. Also, programs that consume large amounts of CPU time will benefit even from modest amounts of enabling. Programs that do small amounts of array manipulation probably will benefit little from the enabling.

The third question helps assess the cost of the enabling. Vector- and parallel-enabled programs are not intrinsically more difficult or expensive to write than non-enabled programs. Vector- and parallel-literate programmers can write an enabled program for a lower cost than less literate programmers.

The fourth question helps assess whether programmer education is one benefit. If programmer education is one benefit, then actual CPU time and elapsed time savings may be lower for the project and still have the project benefits outweigh the project cost.

The last question helps assess whether you can or cannot get results without the enabling. If you cannot get results without the enabling, then you must enable your program.

Rules-Of-Thumb

I noted rules-of-thumb for vector- and parallel-enabling on the IBM 3090. These include:

1. Currently, only FORTRAN programs vector-enable.
2. Only DO loops vector-enable.
3. Loops with CALL statements, I/O statements, or transfers of control do not vector-enable.
4. Loops with recurrences do not vector-enable. One can identify potential recurrences by looking for the same variable on different sides of equals signs on different statements within a loop. Note this last statement is a rule-of-thumb; its truth does not guarantee that a recurrence exists.

5. Loops with small index ranges do not execute faster on a vector processor. Therefore, the vector-enabling compiler will not vector-enable these loops. In loops with variable indices, subscript bounds for arrays referenced inside the loop can clue the programmer about the index range for the loop.

Removing Enabling Inhibitors

I also understand ways to remove inhibitors to vector- and parallel-enabling. These include:

1. Split loops that contain computational code and I/O statements. Place computational code in one loop, and I/O statements in a different loop.
2. Within loops, remove GOTO statements from IF statements. GOTO statements do not vector-enable. For example, convert:

```
       DO 20, I=1,100
           IF (A(I) .EQ. 0.0)  GOTO 10
           B(I) = A(I)
   10      CONTINUE
   20      CONTINUE
```

   into:

```
       DO 20, I=1,100
           IF (A(I) .NE. 0.0)  B(I) = A(I)
   20      CONTINUE
```

3. Remove recurrences. To do this, first define an array for storing temporary results. Next, split loops that contain recurrences into multiple loops. In the first loop, compute intermediate results and store them in the array. In successive loops, read the intermediate results from the array. Once you have removed the recurrence, you may be able to improve the vector performance by introducing vector temporaries and recombining the loops.
4. Restructure the program's solution algorithm to increase its vector content.

Future Developments For AKCESS

Although I have completed much vector-enabling work, many enhancements could be added. The SLINEG vector-enabling could be completed. Double line/double plane and multiple-refinement mesh support could be added to SLINEG. Finally, the vector-enabled SLINEG subroutine could be ported easily to the SLINEJ subroutine.

In the solution algorithm, sweep line computations are independent of the other sweep lines. Hence, a parallel loop could be added around the SLINEG, SLINEJ, and ESOLVE subroutines to process different sweep lines concurrently.

Future Developments For The Benchmark Program

A close look at Table 3 shows a reduction in CPU time as "n" increases. This reduction contradicts the rule that parallel processing reduces the elapsed time required to execute a program, but does not reduce the CPU time. Future work could establish the reason for this reduction. Other future work could execute this benchmark on other processors in the IBM 3090 family, compile and execute it with the parallel processing support in VS FORTRAN Version 2 Release 5, Modification Level 0, and compile and execute it on other supercomputing platforms.

Conclusions

During my research, I concluded several things about supercomputing, vector processing, and parallel processing. First, although today's supercomputers will be tomorrow's workstations, a class of problems solvable only on the latest supercomputer will always exist. Second, the automatic vector- and parallel-enabling of "dusty-deck" programs often is ineffective. Dusty-deck programs must be vector- and parallel-enabled via manual intervention. Third, new engineering and scientific programs should be written to exploit vector and parallel processing, but few programmers understand vector and parallel processing. Therefore, numerical analysis and computational mathematics classes should teach vector- and parallel-enabling. Fourth, parallel processing will become more important to science and engineering as workstation-class equipment offers economical parallel-processing capabilities. Fifth, parallel processing algorithms that contain either very course-grained parallelism or very fine-grained parallelism are the easiest to identify the parallelism within. For example, for course-grained parallelism with 1 to 12 CPUs, the parallelism may be written in parallel subroutines. For very fine-grained parallelism with thousands of CPUs, the parallelism may be written so each computation of a data array manipulation gets executed on a separate CPU. Finally, Amdahl's Law constrains the speedup achievable via vector and parallel processing. Therefore, supercomputers can always use faster semiconductor technologies.

Works Consulted

1. Seemann, Danae, ed. *Abstracts: Research on the Cornell National Supercomputer Facility.* Ithaca, NY: Cornell Theory Center, 1990.

2. Sutherland, J. K. B., ed. *Computer Abstracts,* vol. 34, nos. 1-9. St. Helier, Jersey, British Channel Islands: Technical Information Company, LTD., 1990.

3. Engineering Information, Inc. *The Engineering Index Monthly,* vol. 28, nos. 1-12. New York: Engineering Information, Inc., 1990.

4. Levine, Michael J. Foreword. *Projects in Scientific Computing.* By the Pittsburgh National Supercomputing Center. Pittsburgh: Pittsburgh Supercomputing Center, 1987.

5. Martin, Joanne L. "An Invitation to Participate." *The International Journal of Supercomputer Applications,* vol. 1, no. 1, 1987: p. 3.

6. Smarr, Larry. "Statement from the Director." *Annual Report To The National Science Foundation.* By the National Center for Supercomputing Applications. Champaign, IL: National Center for Supercomputing Applications, 1987.

7. Bloch, Erich. "Supercomputing and the Growth of Computational Science in the National Science Foundation." *The International Journal of Supercomputer Applications,* vol. 1, no. 1, 1987: pp. 5-8.

8. Wayne Schiebel, (Supercomputer Coordinator and Computer Systems Branch Team Leader for Strategic Trade Specialist, U.S. Department of Commerce.) Telephone conversation. June 13, 1991.

9. Doerr, Helen M., and Verdier, Francesca. *Introduction to Vectorization.* Ithaca, NY: Cornell Theory Center, 1989.

10. Soll, David B. *Vectorization and Vector Migration Techniques,* publication number SR20-4966-0. The IBM Corporation, 1986.

11. Doerr, Helen M., and Verdier, Francesca. *Improving Vector Performance (More Vector Bang).* Ithaca, NY: Cornell Theory Center, 1987.

12. Doerr, Helen M., and Verdier, Francesca. *Vector Assembler Considerations (Squeezing Out The Last Cycle).* Ithaca, NY: Cornell Theory Center, 1987.

13. Rutter, Jack (Online Consultant, psfy@cornellf.tn.cornell.edu). Unpublished electronic mail to the author. February 5, 1991.

14. Liu, Yili (NCSA Consulting, consult@ncsa.uiuc.edu). Unpublished electronic mail to the author. February 5, 1991.

15. Hartonas-Garmhausen, Vicky (PSC User Services, remarks@cpwsca.psc.edu). Unpublished electronic mail to the author. February 6, 1991.

16. The IBM Corporation. *3090 Processor Complex: Functional Characteristics,* publication number SA22-7121-7. The IBM Corporation, 1989.

17. The IBM Corporation. *IBM System/370 Extended Architecture: Principles of Operation,* publication number SA22-7085-1. The IBM Corporation, 1987.

18. Verdier, Francesca. "Vectorization Gives You Free Cycles." *Forefronts,* vol. 3, no. 7. Ithaca, NY: Cornell Theory Center, 1987.

19. The IBM Corporation. *IBM Enterprise System Architecture/370 and System/370: Vector Operations*, publication number SA22-7125-3. The IBM Corporation, 1988.

20. Buchholz. W. "The IBM System/370 vector architecture." *IBM Systems Journal*, vol. 25, no. 1, 1986: pp. 51-62.

21. Quinn, Michael J. *Designing Efficient Algorithms for Parallel Computers.* New York: McGraw-Hill, 1987.

22. The IBM Corporation. *VS FORTRAN Version 2 Programming Guide Release 3*, publication number SC26-4222-3. The IBM Corporation, 1988.

23. The IBM Corporation. *Parallel FORTRAN: Language and Library Reference*, publication number SC23-0431-0. The IBM Corporation, 1988.

24. The IBM Corporation. *IBM FORTRAN Translation Tool: Program Description/Operation Manual*, publication number SH20-9256-1. The IBM Corporation, 1988.

25. The IBM Corporation. *VS FORTRAN Version 2: Interactive Debug Guide and Reference Release 3*, publication number SC26-4223-2. The IBM Corporation, 1988.

26. The IBM Corporation. *IBM C/370 User's Guide Release* 2, publication number SC09-1264-03. The IBM Corporation, 1990.

27. Pottle, Marcia. *PTOOL: A Parallel Programming Tool.* Ithaca, NY: Cornell Theory Center, 1988.

28. Allen, Randy, et al. *PTOOL: A Semi-automatic Parallel Programming Assistant*, publication number Rice COMP TR86-31. Houston, TX: Rice University, 1986.

29. The IBM Corporation. *IBM 3090 Vector Facility Simulator Program Description/Operations Manual,* publication number SC23-0336-1. The IBM Corporation, 1986.

30. The IBM Corporation. *Engineering and Scientific Subroutine Library: Guide and Reference,* publication number SC23-0184-2. The IBM Corporation, 1987.

31. Amdahl, Gene. "The Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities." *AFIPS Conference Proceedings Spring Joint Computer Conference,* 30, 1967: pp. 483-485.

32. Clark, R. S., and Wilson, T. L. "Vector system performance of the IBM 3090." *IBM Systems Journal,* vol. 25, no. 1, 1986: pp. 63-82.

Appendixes

Appendix A. Supercomputing Facilities

This appendix summarizes information about the supercomputing facilities available to researchers at the University of Tennessee, Knoxville (UTK). It describes the facilities available both in Knoxville and via UTK's affiliations with national supercomputing centers.

The University Of Tennessee Computing Center

The University of Tennessee Computing Center (UTCC) has an IBM 3090-300E computer. It has a 32-bit word, 128 megabytes (Mb) of central storage, and 128 Mb of expanded storage. It contains 3 central processing units (CPUs) and 3 integrated vector facilities (VFs). Its estimated per-processor performance is 15 million instructions per second (MIPS) or 116 million floating-point operations per second (MFLOPS). The IBM 3090 300E at UTCC executes the MVS/SP operating system.

The Cornell National Supercomputing Facility

The Cornell National Supercomputing Facility (CNSF) has two IBM 3090-600J computers. [13] Each has a 32-bit word, 512 Mb of central storage, and 1 gigabyte (Gb) of expanded storage. Each contains 6 CPUs and 6 VFs. The estimated per-processor performance of each machine is 18 MIPS or 138 MFLOPS. Both IBM 3090 600Js at the CNSF execute the VM/XA/SP operating system.

Currently, CNSF users can execute parallel programs that use all 6 CPU/VFs on a single machine. The CNSF is interconnecting the two IBM 3090s so its users can execute parallel programs that use all 12 CPU/VFs on the two machines.

The National Center For Supercomputing Applications

The National Center for Supercomputing Applications (NCSA) has two computers: a Cray 2 and a Cray Y/MP. [14] Both machines have 64-bit words. The Cray 2 has 128 megawords (Mw) of central storage. The Cray Y/MP has 64 Mw of central storage and 128 Mw of solid-state disk (SSD). Each machine has 4 CPUs with 4 integrated vector processors. The estimated performance of the Cray 2 is 0.8 to 1.2 billion floating point operations per second (GFLOPS). The estimated performance of the Cray Y/MP is 1.2 GFLOPS. Both execute the Unicos operating system.

The Pittsburgh Supercomputing Center

The Pittsburgh Supercomputing Center (PSC) has two computers: a Cray Y/MP and a Thinking Machines Corporation model CM-2 Connection Machine. [15] The Cray Y/MP has a 64-bit word, 32 Mw of central storage, and 128 Mw of SSD. It has 8 CPUs with 8 integrated vector processors. The Cray Y/MP executes the Unicos operating system. The Connection Machine has 256 kilobits of storage per processor. It has 32,768 bit-serial processors and no vector processors.

Appendix B. Architecture Of The IBM 3090

One can view the IBM 3090 at many levels. This appendix describes the computer and CPU views of the IBM 3090. The computer view includes the central processing unit (CPU), storage (memory) and I/O channels. This view does not include I/O devices. The CPU view shows a block diagram of the CPU only. This appendix also discusses the programming model of the IBM 3090 and lists the datatypes the IBM 3090 supports. In this appendix, speed and capacity specifications describe the E model series of the IBM 3090 processor family.

Machine Architecture

The IBM 3090 computer has one or more CPUs, the storage subsystem, the I/O channel subsystem, and the system control element. [16] Together, these subsystems correspond to the motherboard in a microcomputer system. Figure 4 shows a block diagram of these subsystems.

CPU

The model 300E has 3 CPUs. Each executes machine instructions and may include an optional vector facility (VF). IBM manufactures the CPUs from emitter-coupled logic (ECL) integrated circuits (ICs). The company mounts and interconnects up to 132 ICs together in a single thermal conduction module

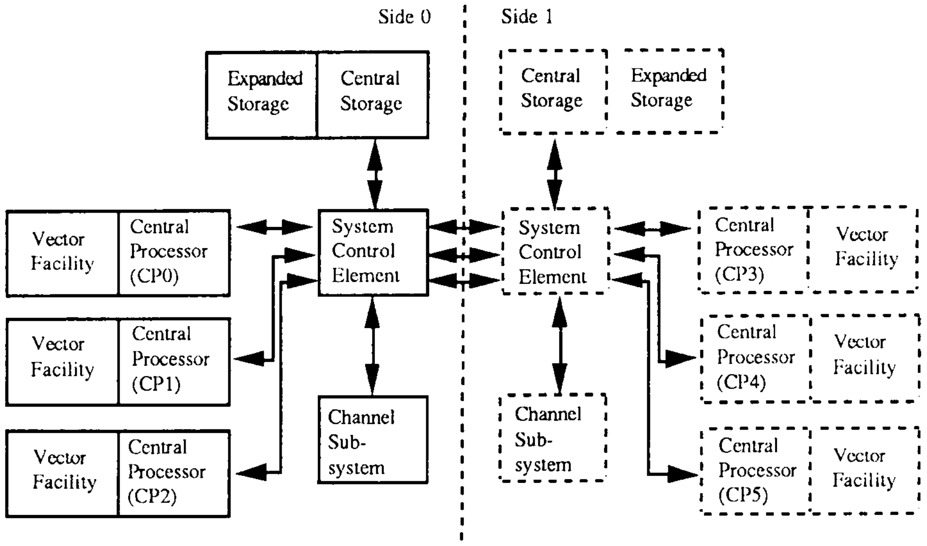


Figure 4. IBM 3090-600 Block Diagram: Source: The IBM Corporation. *3090 Processor Complex Functional Characteristics*, publication number SA22-7121-7. The IBM Corporation, 1989. p. 3-2.

(TCM). IBM interconnects 9 TCMs on a multilayer backplane to form a single CPU. Each CPU has a cycle time of 17.2 nanoseconds and can execute approximately 15 million instructions per second (MIPS).

Storage Subsystem

Central storage and expanded storage comprise the storage subsystem. Semiconductor random-access memory (RAM) comprises both central and expanded storage. The CPUs fetch instructions and operands from central storage. Expanded storage improves the machine's paging performance by reducing its virtual storage paging to disk.

All CPUs in a model 300E share central storage. A model 300E can have either 64 megabytes (Mb) or 128 Mb of central storage. Central storage is byte-addressable, but most instructions reference storage on 4-byte (word) boundaries. Data transfers between central storage and the processor cache occur 128 bytes at a time. For these transfers, the first 8-byte (1 doubleword) fetch requires 22 machine cycles. Subsequent doubleword fetches require 1 machine cycle.

A model 300E can have 0 Mb to 1024 Mb of expanded storage. The CPUs cannot fetch instructions and operands directly from expanded storage; instead they transfer virtual storage pages between central storage and expanded storage. Data transfers to or from expanded storage pass through central storage. These data transfers occur 1 page (4096 bytes) at a time and require 1 machine cycle per byte.

I/O Channel Subsystem

I/O channels are intelligent processors that receive and execute commands to access I/O devices. Each operates independently of the CPU and other I/O channels. They return the data, status information, or error information in central storage. The model 300E comes equipped with 32 I/O channels; customers optionally can equip it with 40, 48, or 64 channels. The I/O channels, with a single channel control element, comprise the I/O channel subsystem.

The System Control Element

The system control element switches data between central storage, the I/O channel subsystem, and the CPUs. It ranks storage access requests and performs error-checking and reporting. It also interrogates the processor caches to insure that storage fetches return the most recent copy of a storage location.

CPU Architecture

The IBM 3090 CPU consists of the control storage element (CSE), the instruction element (IE), the execution element (EE), and the buffer control element (BCE). [16] The CPU may include the optional vector facility (VF). Figure 5 shows a block diagram of the IBM 3090 CPU.

The CSE contains the control registers and the machine microcode. It fetches the microinstructions that control the IE and EE and also controls the microcode execution.

The IE decodes machine instructions, controls the sequencing of the machine instructions, calculates storage addresses, and sends storage fetch requests to the BCE. Finally, it provides opcodes, operands, and operand addresses to the EE.

The EE executes machine instructions and operates in parallel with the IE. It also processes interruptions, executes logical decisions, and executes arithmetic and control functions.

The BCE processes all CPU references to central storage. It performs the dynamic address translation required to implement virtual storage. It also contains 64 kilobytes (kb) of high-speed storage cache.

The optional VF extends the instruction and execution units. It adds 63 vector instructions and 171 opcodes to the machine instruction set. Appendix C discusses the vector facility in more detail.

Programming Model

The programming model of the IBM 3090 includes a program status word, general-purpose registers, floating point registers, and control registers. [17] Figure 6 shows the IBM 3090 programming model.

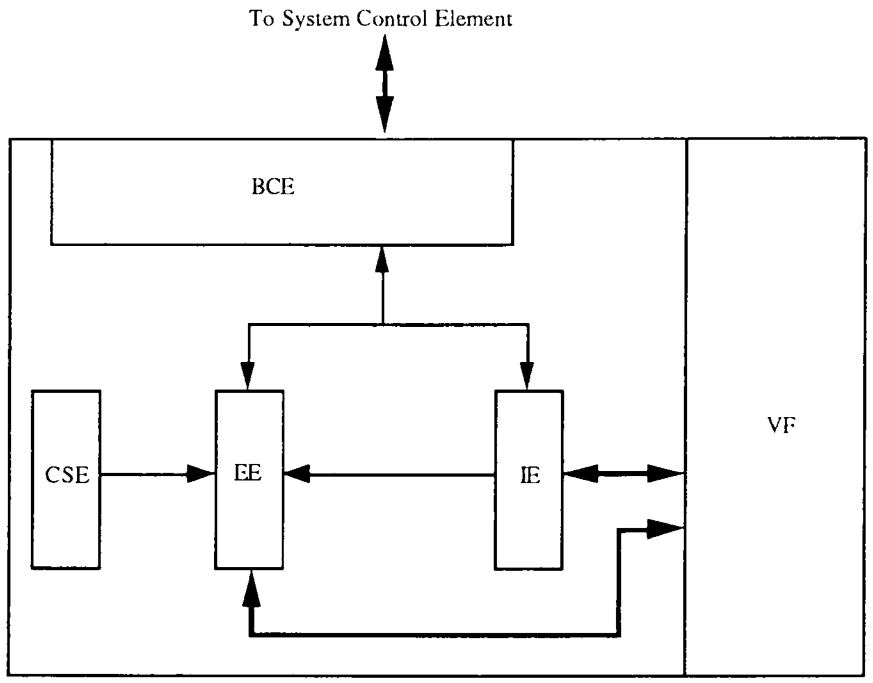
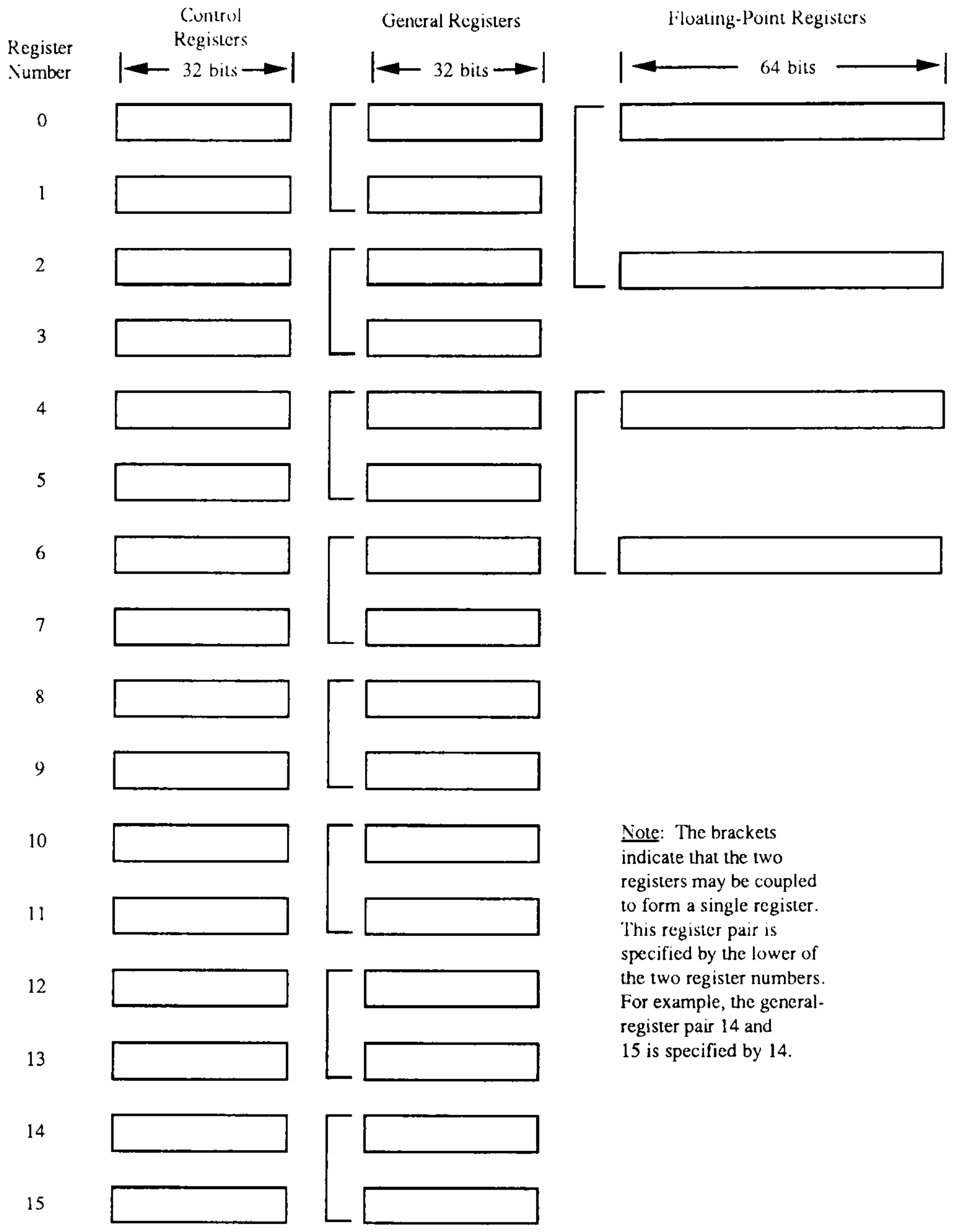


Figure 5. IBM 3090 CPU Block Diagram: Source: The IBM Corporation. *3090 Processor Complex Functional Characteristics,* publication number SA22-7121-7. The IBM Corporation, 1989. p. 3-4.

The program status word is 64 bits wide. It contains the program counter, condition codes resulting from arithmetic comparisons, and interrupt mask information. It also contains other CPU state and control information.

Programmers use the general-purpose registers to store operands, accumulate results, or as base and index registers for storage address calculations. The general-purpose registers are 32 bits wide, but may be paired to form fewer 64-bit wide registers.

Programmers use the floating point registers to store floating-point operands and results. The floating point registers are 64 bits wide, but can be paired to form fewer 128-bit wide registers.



Register Number
Control Registers
32 bits
General Registers
32 bits
Floating-Point Registers
64 bits
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Note: The brackets indicate that the two registers may be coupled to form a single register. This register pair is specified by the lower of the two register numbers. For example, the general-register pair 14 and 15 is specified by 14.

Figure 6. IBM 3090 CPU Programming Model: Source: The IBM Corporation. *IBM System/370 Extended Architecture Principles of Operation,* publication number SA22-7085-1. The IBM Corporation, 1987. p. 2-4.

Programmers cannot directly access the control registers. Bit positions in these registers indicate installed hardware facilities or valid control operations. The control registers are 32 bits wide.

Supported Data Types

The IBM 3090 supports:

- 32-bit unsigned binary numbers;
- 32-bit signed binary numbers;
- 32-bit floating-point numbers;
- 64-bit signed binary numbers;
- 64-bit floating-point numbers;
- 128-bit floating-point numbers;
- 16-byte decimal numbers. [17]

For 16-byte decimal numbers, each decimal digit requires one nybble (4 bits). This datatype consists of 31 digits and a sign nybble.

Appendix C. Vector Processing And The IBM 3090

This appendix explains data vectors, vector machine instructions, pipelined machine execution, the performance of an execution pipeline, and vector sectioning. It also describes the IBM 3090 vector facility (VF) and its programming model. Finally, this appendix lists the data types supported by the VF.

Data Vectors And Vector Machine Instructions

A scalar data element is a single data value or operand. A data vector is a set of scalar data elements that all have the same data type. The vector length is the number of scalar elements in the data vector. The vector stride is the number of storage bytes between each element of the data vector. Programmers access individual elements of a data vector with either high-level language subscripts or assembly-language indirect addresses. The variable A is an example of scalar data; the array A(1), A(2), A(3), . . . is an example of vector data.

A vector machine instruction performs the same arithmetic operation on each element of a data vector. Vector instructions execute faster than scalar instructions on a per-scalar-element basis. This occurs partly because a vector processor fetches fewer instructions per element than a scalar processor. Thus, the vector processor accesses storage fewer times per element and conserves storage bandwidth. Pipelined execution provides most of the vector processor performance increase.

Pipelined Execution Of Numeric Calculations

One can divide numeric calculations into distinct sub-operations. For example, one can divide floating-point multiplication into pre-normalization, mantissa multiplication, exponent addition, and post-normalization sub-operations. [18] Vector processors have hardware stages for executing each sub-operation. In these stages, each sub-operation requires the same amount of time. Each stage operates independently of, and concurrently with, the other stages. The stages are connected end-to-end, so results from one stage can be transferred to the next stage. In a sense, calculations "flow" through the hardware stages like water flows through a pipeline. This manner of operation is called pipelined execution. As results leave the pipeline, new operands enter the pipeline. Thus, four calculations, in various stages of completion, can execute concurrently. Figure 7 illustrates how pipelined execution works.

Pipeline Performance

For the hypothetical four-stage pipeline described above, results from the first calculation appear at the output four machine cycles after entering the pipeline. Subsequent results appear at the pipeline output after each machine cycle.

The pipeline requires time for initialization; thus, it has a start-up delay. The start-up delay is amortized over the number of calculations. As the vector length increases, given a constant start-up delay, the calculation rate approaches one calculation per machine cycle.

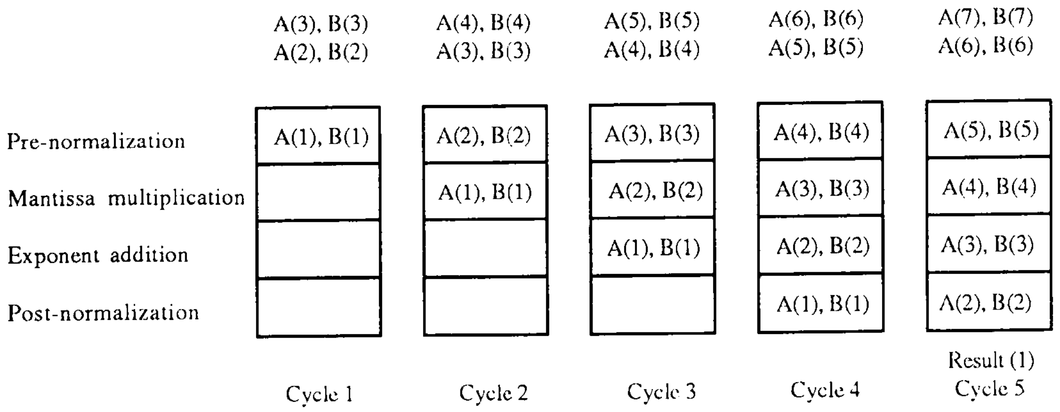


Figure 7. Pipelined Execution: Source: Verdier, Francesca. "Vectorization Gives You Free Cycles." *Forefronts,* vol. 3, no. 7, Ithaca, NY: Cornell Theory Center, 1987.

Because of the start-up delay, an operation with a vector of a certain length executes equally fast on the scalar processor as it executes on the vector processor. This vector length is known as the break-even length and depends on the vector operation. An operation with data vectors shorter than the break-even length execute slower on the vector processor than on the scalar processor. An operation with data vectors longer than the break-even length execute faster on the vector processor than on the scalar processor.

Vector Sectioning

Because of implementation limitations, a single vector instruction can process a vector of some maximum length. This maximum length is called the vector section size and depends upon the machine. The vector processor operates on vectors longer than the section size in chunks. Vectors longer than the section size must be split into chunks, each less than or equal to the vector section size. This splitting is known as sectioning. The vector processor sequentially processes the vector sections until the entire vector has been processed.

High-level language compilers automatically section data vectors. Assembly language programmers must manually section data vectors. Software section-control loops control the execution of the vector sections. High-level language compilers automatically generate these, but assembly language programmers must manually write them. Figure 8 shows a FORTRAN sectioning example.

The IBM 3090 Vector Facility

The vector facility (VF) optionally may be added to the IBM 3090 CPU. The VF in an IBM 3090 CPU is analogous to an 8087 coprocessor in an 8086-based system. It extends the CPU instruction set by 63 vector instructions and 171 opcodes. [19]

The break-even length for the VF is 12 to 20 elements, depending upon the vector instruction. The VF section size is 128 elements. For the multiply and add, multiply and subtract, and multiply and accumulate instructions, the VF can execute two floating-point operations per machine cycle. For these instructions, its theoretical peak performance is 116 million floating-point operations per second (MFLOPS).

Programming Model

The programming model for the IBM 3090 VF includes vector registers, a vector mask register, a vector status register, and a vector activity count register. [20] Figure 9 shows the VF programming model.

```
      DO 10,J=1,N
10       A(J)=B(J)
```

Before Sectioning

```
           DO 10,J=1, N, Z

        DO xx,jv=J, J+MIN(N-J,Z-1), 1
xx          A(jv) = B(jv)

10      CONTINUE
```

After Sectioning

Figure 8. Vector Sectioning: Source: Soll, David B. *Vectorization and Vector Migration Techniques*, publication number SR20-4966-0. The IBM Corporation, 1986. p. 20.

The 16 vector registers temporarily store the vector operands of arithmetic, comparison, and logical vector operations. Each scalar element in a vector register is 32 bits wide, but the vector registers can be paired to hold 64-bit data elements.

The vector mask register is the target for vector comparison instructions. It is also the source and target for logical operation on bit vectors and is the mask source for mask controlled vector operations. The vector mask register has a 1-bit scalar element for each scalar element in a vector register.

The contents of the vector status register describe the status of the vector register, the vector mask register, and the mode of operation. The vector status register has one 64-bit scalar element.

The vector activity count register helps measure the CPU time consumed executing vector instructions. The system clock increments this register as the VF executes. The vector activity count register has one 64-bit scalar element.

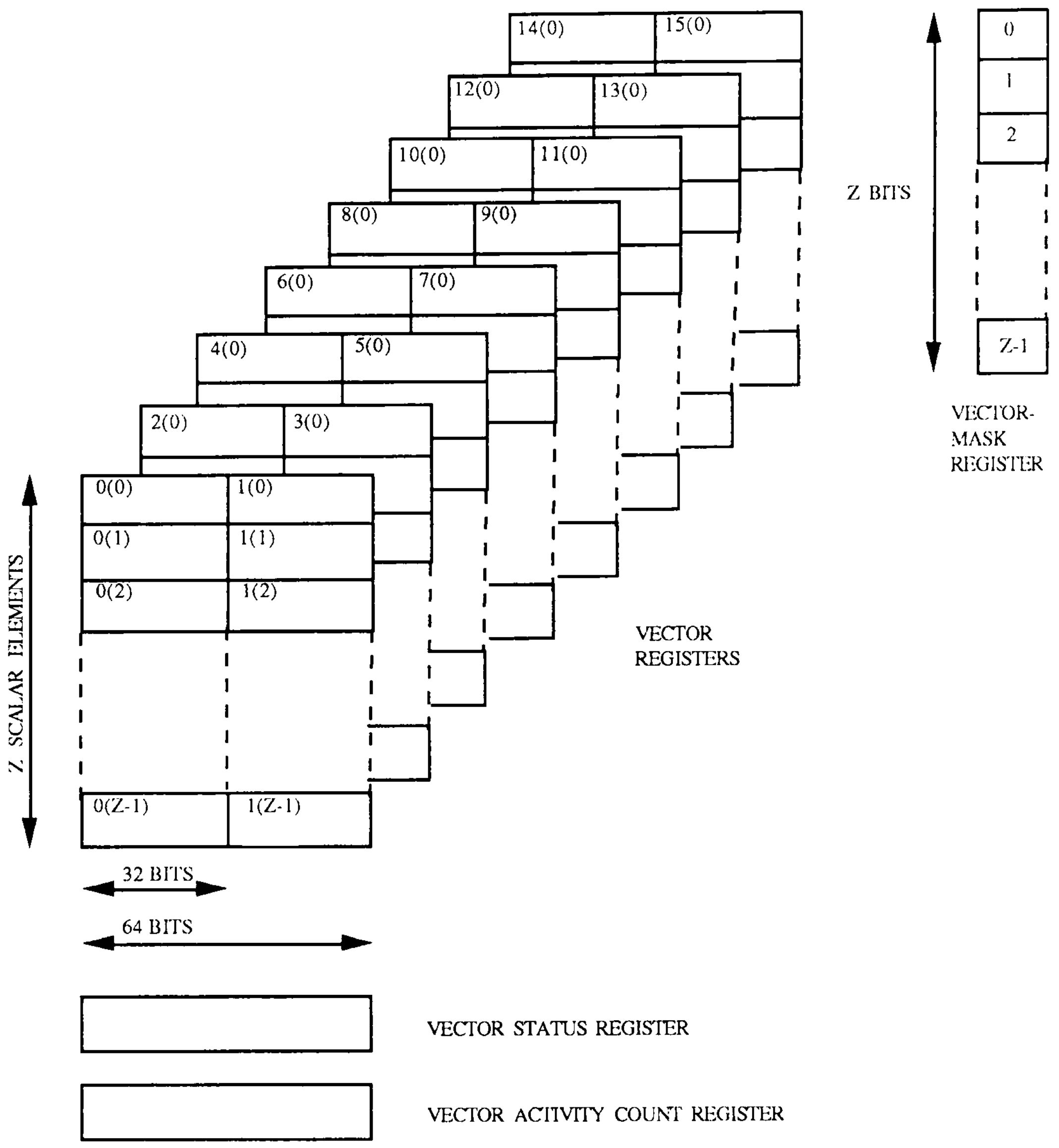


Figure 9. Vector Facility Programming Model: Source: Buchholz, W. "The IBM System/370 Vector Architecture." *IBM Systems Journal*, vol. 25, no. 1, 1986: p. 53.

Data Types

The VF supports:

- 1-bit logical data;
- 16-bit signed binary numbers;
- 32-bit signed binary numbers;
- 32-bit floating point numbers;
- 64-bit signed binary numbers;
- 64-bit floating point numbers. [19]

Appendix D. Parallel Processing And The IBM 3090

This appendix introduces parallel processing, its benefits, its operation, and its limitations. This appendix also discusses the IBM 3090 facilities for parallel processing.

Overview

Parallel processing partitions a single problem into computationally-independent pieces, then executes the pieces on multiple central processing units (CPUs). This lets parallel-enabled programs execute more computations per unit time than non-parallel-enabled programs. Parallel-enabled programs can exceed the computation rate barrier imposed by the physical limits of the CPU architecture, logic implementation, and cost of developing a substantially faster single CPU.

Requirements

Algorithms, hardware, and software combine to implement parallel processing. Omitting any of these will inhibit parallel processing.

Parallel processing requires an algorithm with computationally-independent calculations. The operands and results of computationally-independent calculations do not depend upon or affect other calculations. Each calculation can execute at any time without regard to other calculations. Consequently, these calculations can execute concurrently.

Parallel processing also requires computer hardware support. It requires multiple interconnected CPUs. Separate pieces of a single program cannot concurrently execute on a single CPU. The multiple CPUs must have interconnecting communications paths to share operands and results.

Finally, the operating system and utilities must support parallel processing. Language compilers must be able to translate parallel work into machine instructions and operating system calls. The operating system must be able to load and execute the parallel program.

Flynn's Taxonomy

Computer architects classify parallel processors based on the numbers of instruction streams and data streams the processors support. [21] Instruction streams are series of machine instructions grouped for execution by a single CPU. Data streams are sets of operands grouped for manipulation by a single CPU.

Single Instruction stream, Single Data stream (SISD) machines have a single instruction decoder and a single execution unit. Single processor computers exemplify this class of machine. Single Instruction stream, Multiple Data stream (SIMD) machines have a single instruction decoder and multiple execution units. The Connection Machine (from Thinking Machines Corporation) exemplifies this class of machine. Multiple Instruction stream, Single Data stream (MISD) machines have multiple instruction decoders and a single execution unit. No machines ever built exemplify this class of machine. Multiple Instruction stream, Multiple Data stream (MIMD) machines have multiple instruction decoders and multiple execution units. Multiple-CPU computers such as Crays, IBM 3090s, and Alliants exemplify this class of machine.

Interprocessor Communications

Computer architects have devised two schemes for communicating between CPUs: shared storage and message passing. In shared storage machines, all CPUs can access the same storage locations. In message passing machines, each CPU has local storage separate from the others. The CPUs communicate via messages passed across communications paths between the CPUs.

In shared storage machines, any CPU can store into a shared storage location. The other CPUs then can fetch the contents of that shared storage location. In shared storage machines, the incorrect sequencing of shared storage accesses causes hard-to-find programming errors. This type of machine requires careful arbitration and sequencing of storage accesses to prevent incorrect results.

In message passing machines, each CPU has local storage separate from the others. The CPUs communicate via messages passed across communications paths between the CPUs. For these machines, computer architects have devised many different configurations for interconnecting the CPUs. These configurations include switch networks, trees, pyramids, hypercubes, and others.

Limitations

Parallel processing reduces the elapsed time required to execute a program; it does not reduce the CPU time required. Amdahl's Law establishes a theoretical limit for the reduction in elapsed time achievable via parallel processing. Appendix F describes Amdahl's Law in more detail.

Besides the theoretical limit imposed by Amdahl's Law, parallel processing has practical limitations. These limitations include the availability of programming tools that can exploit parallelism in existing programs and the availability of CPU time on the parallel processor.

Automatic parallel-enabling tools have limitations. The tools cannot exploit parallelism obscured by poor program coding. Also, the tools cannot infer the program's underlying algorithm and either exploit its inherent computational independence or select an algorithm with greater computational independence. For these reasons, programmers must manually parallel-enable programs.

Finally, parallel-enabled programs cannot reduce the elapsed time of a program unless the parallel processor has unused CPU time. The parallel program will not benefit from execution on a fully-utilized parallel machine.

The IBM 3090 Facilities For Parallel Processing

IBM manufactures and sells 3090 computers with 1 to 6 CPUs and vector facilities (VFs). The multiple-CPU machines are MIMD-class machines. In the multiple-CPU machines, interprocessor communication occurs through shared central storage. Figure 4 on page 51 shows the architecture of the IBM 3090.

Parallel processing software includes the PTOOL data dependence analyzer, the VS FORTRAN Version 2 compiler, the Parallel FORTRAN Prototype compiler, and the C/370 Release 2 compiler. IBM's MVS/SP and VM/XA operating systems support parallel processing. Appendix E describes these software tools in more detail.

Appendix E. Software Tools

This appendix surveys software tools for vector-enabling and parallel-enabling programs for execution on the IBM 3090. It discusses the VS FORTRAN Version 2 compiler, the Parallel FORTRAN Prototype compiler, the FORTRAN Translation Tool, and the VS FORTRAN Interactive Debugger (IAD). It also discusses the C/370 Release 2 compiler, the PTOOL data dependence analyzer, the Vector Facility Simulator (VSIM), and the Engineering and Scientific Subroutine Library (ESSL).

The VS FORTRAN Version 2 Compiler

The VS FORTRAN Version 2 compiler translates source programs into object code. [22] Its vector- and parallel-enabling features include automatic vector-enabling of DO loops, diagnostic messages about vector-enabling inhibitors, and the Multi-Tasking Facility (MTF). Additionally, Release 5 includes parallel programming extensions previously available only in the Parallel FORTRAN Prototype compiler. This compiler's limitations include nonstandard language extensions for the MTF and parallel programming, support for only coarse-grained parallelism with the MTF, and support for the MTF only on the MVS/SP operating system. A programmer would use this compiler for general-purpose and vector-enabled FORTRAN program development. He or she would not use the MTF or parallel programming extensions in portable programs.

The IBM Parallel FORTRAN Prototype Compiler

The IBM Parallel FORTRAN Prototype compiler translates source programs into object code. [23] Its vector- and parallel-enabling features include automatic vector- and parallel-enabling of DO loops, diagnostic messages about enabling inhibitors, language extensions for parallel programming, parallel event tracing, and parallel program execution support on the MVS/SP and VM/XA operating systems. The language extensions support explicit parallelism including parallel task management, storage locks, communication between parallel tasks, and parallel event scheduling.

This compiler's limitations include nonstandard language extensions for parallel programming, restricted availability from IBM, and VS FORTRAN Version 2 Release 1 as a base for the compiler. A programmer would use this compiler to develop vector- and parallel-enabled FORTRAN programs. He or she would not use the parallel programming extensions in portable programs.

The FORTRAN Translation Tool

The FORTRAN Translation Tool automatically converts DEC and CDC dialects of FORTRAN to FORTRAN 77. [24] It converts CDC FORTRAN Versions 2.3, 4, or 5, DEC PDP-11 FORTRAN IV and FORTRAN IV Plus, and DEC TOPS 10/20 FORTRAN IV (FORTRAN 10 and FORTRAN 20). It also converts DEC VAX-11 FORTRAN Version 3.0 and DEC VAX FORTRAN Version 4.0. This tool's limitations include non-fully-automatic conversion; manual modifications may be required. A programmer would use this tool to port FORTRAN programs from source FORTRAN dialects to VS FORTRAN.

The VS FORTRAN Interactive Debugger

The VS FORTRAN Interactive Debugger (IAD) helps debug FORTRAN programs, locate execution hot spots, and gather vector length and vector stride information. [25] A programmer would use the IAD to debug programs and to gather performance and tuning information.

The IBM C/370 Release 2 Compiler

The IBM C/370 Release 2 compiler translates C source programs into object code. [26] This compiler's vector- and parallel-enabling features include its Multi-Tasking Facility (MTF). This compiler's limitations include a lack of automatic vector- or parallel-enabling, nonstandard language constructs for the MTF, and MTF execution support only on the MVS operating system. A programmer would use this compiler for general-purpose C program development. He or she would not use the MTF in portable programs.

The PTOOL Data Dependence Analyzer

PTOOL analyzes FORTRAN source code, identifies inhibitors to parallel-enabling, and identifies variables that should reside in shared storage. [27] [28] Its vector- and parallel-enabling features include automatic analysis, non-system-specific results, a more sophisticated analysis than the Parallel FORTRAN Prototype compiler provides, and reports about its reasoning. Its limitations include a static analysis only, analysis of FORTRAN programs only, analysis of DO loops and backward GOTOs only, and an inability to analyze parallel constructs. A programmer would use PTOOL to identify inhibitors to vector- and parallel-enabling. He or she also would use PTOOL to learn about data dependencies.

The Vector Facility Simulator

The Vector Facility Simulator (VSIM) simulates the execution of IBM 3090 vector machine instructions. [29] Its vector- and parallel-enabling features include its abilities to execute vector-enabled programs on non-VF-equipped machines, to gather vector length and stride information, to provide the same results as a VF, and to access the simulated vector register contents easily. Its limitations include software simulation of vector machine instruction execution. This slows its execution and limits its usefulness for executing production programs. A programmer would use VSIM to execute a vector-enabled program on a non-VF-equipped machine. He or she also would use VSIM to gather vector length and stride information for program tuning.

The Engineering And Scientific Subroutine Library

The Engineering and Scientific Subroutine Library (ESSL) has vector-enabled subroutines for common engineering and scientific calculations. [30] The ESSL has more than 230 linear algebra, matrix algebra, eigensystem analysis, signal processing, sorting and searching, interpolation, numerical quadrature, and random number generation subroutines. Its vector- and parallel-enabling features include vector-enabled subroutines tuned for the maximum performance on an IBM 3090 VF, scalar subroutines for program development, Basic Linear Algebra Subroutine (BLAS) interfaces to the subroutines, support for multiple datatypes, and support for calling from VS FORTRAN, C/370, or S/370 assembly language. Its limitations include proprietary calling interfaces, and non-bitwise-identical results returned from the vector and scalar versions of the same subroutine. A programmer would use ESSL to vector-enable a program and get the maximum program performance with the least effort. He or she would not use ESSL to develop portable programs.

Appendix F. Amdahl's Law

This appendix states and develops Amdahl's Law. It also contains observations about Amdahl's Law and its implications for supercomputing.

Overview

Amdahl's Law relates the overall program speedup as a function of two variables. The first is the program fraction that can be executed on a vector processor. The second variable is the vector/scalar speed ratio of the computer. Explicitly, Amdahl's Law [31] [32] is:

```
                     1
s(f,r) =  -----------
          (1-f) + f/r
```

where:

```
s = the overall program speedup
f = the vector fraction of the program
r = the vector/scalar speed ratio
```

Development

Figure 10 helps explain this development graphically. First define the variables:

t1	the central processing unit (CPU) time required to execute the unmodified program
t2	the CPU time required to execute a vector-enabled version of the same program
s	program speedup
ta	the scalar execution time component of the program
tb	the fraction of t1 that can be replaced with vector instructions and executed on a vector processor
tb'	the vector-execution time component of t2
r	the ratio of the vector calculation rate and scalar execution rate

```
                  vector calculation rate
          r  =    -----------------------
                  scalar calculation rate
```

f	the fraction of the original program that can be replaced by vector machine instructions and executed on a vector processor. Note that f is the same as tb; this additional symbol has been introduced for clarity during the development
tr	the reduction in CPU time consumed by the vector-enabled program

The objective is to find s(f,r).

```
1.     s  = t1 / t2

2.     t1 = ta + tb

3.     t2 = ta + tb'
```

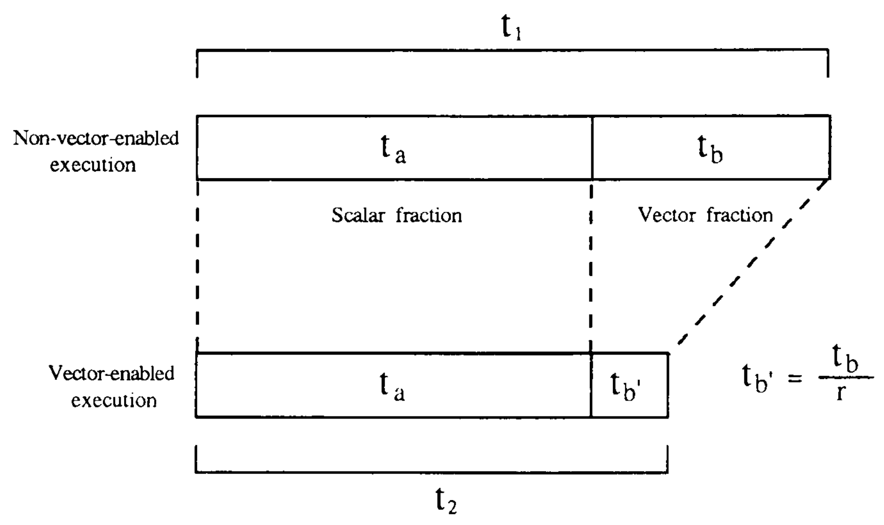


Figure 10. Development Of Amdahl's Law

4. But, tb' = tb / r , where

$$r = \frac{\text{vector calculation rate}}{\text{scalar calculation rate}}$$

5. t2 = ta + (tb / r)

6. $$s = \frac{ta + tb}{ta + (tb/r)}$$

7. Setting t1 = 1 to scale results against unity:

t1 = 1, so ta = (1-tb), and

$$s = \frac{1}{(1-tb) + (tb/r)}$$

8. Noticing that tb is really the vector fraction f

$$s(f,r) = \frac{1}{(1-f) + (f/r)}$$

Reductions In Elapsed Time

The variable s defines the overall program speedup. We can convert s to reductions in CPU time (tr) with the following steps:

```
1.     tr = t1 - t2
2.     We know  s = t1 / t2, therefore  t2 = t1 / s
3.     Thus,  tr = t1 - ( t1 / s )
```

Observations

Figure 11 shows a graph of Amdahl's Law. From this figure, one sees the vector/scalar speed ratio r affects the overall speedup s relatively little until the vector fraction f approaches one. One also sees that programs that have mid-range vector fractions benefit almost as much from modest vector/scalar speed ratios as they benefit from high vector/scalar speed ratios.

This development only considers CPU time consumption; it does not consider I/O wait, storage page wait, or scheduler wait time. Thus, this development and conclusions apply only to programs whose execution is constrained by CPU speed. It does not apply to programs whose execution is constrained by I/O waits, storage page waits, or scheduler waits.

Reflection on this development shows the same relationships hold for parallel/serial speed ratio in a parallel processor and the overall speedup of a parallel-enabled program. From this, we see that Amdahl's Law defines the theoretical maximum speedup available via parallel processing. [21]

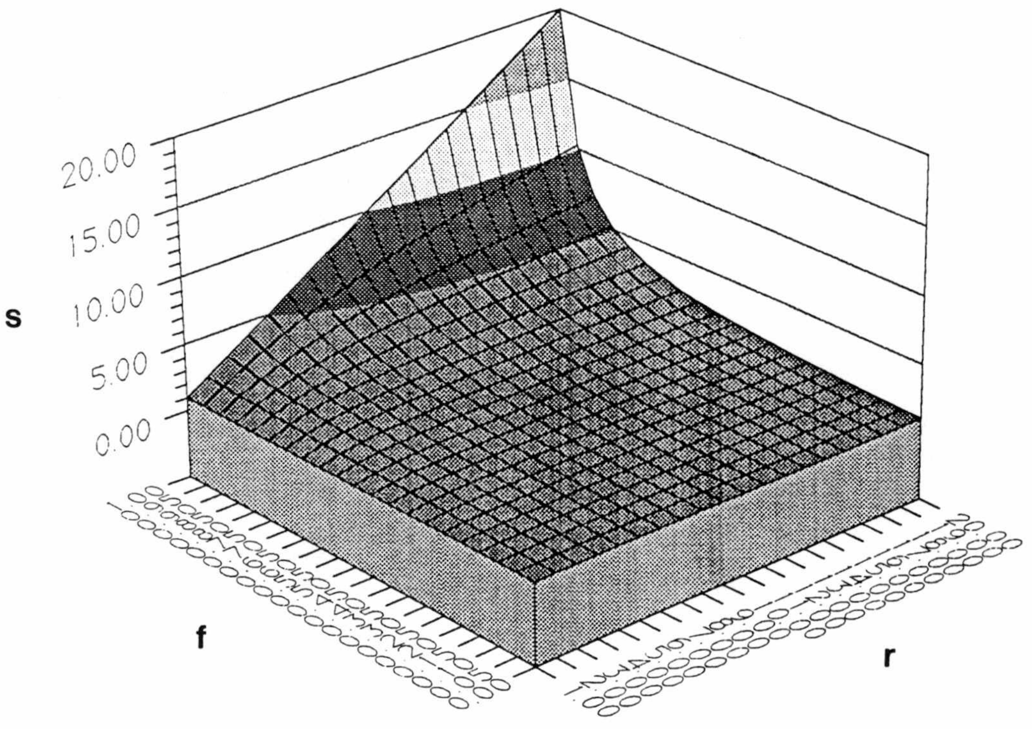


Figure 11. Amdahl's Law

Implications Upon Supercomputing

Amdahl's Law shows that high vector/scalar speed ratios do not guarantee large overall program speedup. The serial fraction of the program constrains the overall program speedup. Thus, to maximize the performance of a program on a vector and parallel processing system, the programmer must minimize the serial scalar-execution fraction of the program. Also, in any vector or parallel processing system, at least one CPU should be extremely fast. [21]

Glossary

Amdahl's Law: A mathematical relationship that defines the theoretical maximum program speedup achievable via vector or parallel processing.

break-even length: The data vector length that executes equally fast on scalar and vector processors.

cache: A memory buffer that operates at CPU speeds.

central storage: Byte-addressable storage from which the IBM 3090 CPUs fetch instructions and operands.

coarse-grained parallelism: Parallelism in which the amount of CPU time consumed by a parallel task is large relative to the CPU time consumed by the entire program.

computational independence: An independent relationship between sets of operands and their corresponding results such that any result can be computed at any time.

computational tractability: The ease with which a problem may be solved on a given computer.

computationally tractable: Solvable on a given computer.

computationally-independent: Not requiring the results of other calculations as input, nor affecting other calculations with its results.

CPU time: The length of time a single program keeps a single CPU busy.

data dependence: A relationship between operands and results that may prevent vector or parallel processing.

data stream: A set of operands grouped for manipulation by a single CPU.

data vector: A set of scalar data values that all have the same data type.

datatype: An interpretation of a binary number.

doubleword: On the IBM 3090, an 8-byte quantity.

dynamic address translation: Mapping a virtual storage address to a real storage address as a CPU references the virtual storage address.

elapsed time: The time difference between when a program starts execution and when it ends execution.

enabling inhibitor: Anything that prevents a programmer from modifying a program to execute on a vector processor or on multiple CPUs in a multiple-CPU system.

expanded storage: Storage that improves the virtual storage paging performance of an IBM 3090.

explicit parallelism: Parallelism explicitly specified by the programmer via parallel-programming language constructs.

hot spot: A section of a program that consumes a disproportionately large amount of CPU time.

I/O channel: An intelligent processor that receives and executes commands to access I/O devices.

I/O wait: A program execution delay caused by an incomplete data transfer between an I/O device and storage.

I/O-constrained: The program's execution rate is limited by the data transfer rate to or from an I/O device.

instruction stream: A series of machine instructions grouped for execution by a single CPU.

language extensions: An additional statement or construct in a programming language that does not conform to a multi-vendor, national, or international definition for that programming language.

MIMD: Multiple Instruction stream, Multiple Data stream.

Multi-Tasking Facility: A set of programming language extensions for explicitly adding, deleting, and managing entries in the operating system scheduler queue.

non-bitwise-identical: Two or more binary numbers that differ in one or more of the least significant bit positions.

non-fully-automatic: Requires manual assistance.

non-parallel-enabled: Not modified to execute concurrently on multiple CPUs in a multiple-CPU system.

non-system-specific: Applicable to more than one computer system.

non-VF-equipped: Not equipped with an IBM 3090 Vector Facility.

numerically-intensive: Requiring many numeric calculations.

nybble: A 4-bit binary number.

object code: The machine instructions generated as output from a language compiler or assembler.

overall program speedup: The difference in execution time between a program and a vector-enabled or parallel-enabled version of the same program.

paging: The operation of transferring one or more virtual storage pages between central storage and expanded storage or between central storage and disk storage.

parallel-enable: To modify a program to execute concurrently on multiple CPUs in a multiple-CPU system.

parallel event: An event that occurs during a parallel-enabled program's execution.

parallel event scheduling: Specifying the execution of, or the execution order of, one or more parallel events.

parallel event tracing: Tracing or logging parallel events.

parallel processing: Concurrently executing a single program on multiple CPUs in a multiple-CPU system.

parallel task management: Creating, deleting, monitoring, or communicating with parallel tasks.

parallel task: A program segment or subroutine that performs a specific function and can execute concurrently with other parallel tasks on a multiple-CPU system.

platform: A computer and operating system.

scalar data element: A single data value or operand.

scalar instruction: An instruction for a single CPU, not for a vector processor.

scheduler wait: A program execution delay caused by the operating system scheduler not allowing a program to start or resume execution.

section: To split a long data vector into shorter pieces so each piece can be processed with a single vector instruction.

serial fraction: The fraction of a program that cannot be executed on a vector processor or concurrently executed on multiple CPUs in a multiple-CPU system.

solid-state disk: A memory device, composed of solid-state RAM, that is accessed like disk storage.

source program: The high-level language program supplied as input to a language compiler.

static analysis: An analysis that is not updated interactively as the program is updated.

storage: Memory.

storage lock: A technique by which parallel tasks serialize access to shared storage locations.

storage page wait: A program execution delay caused by an incomplete virtual storage paging operation.

storage-constrained: Execution rate limited by the availability of virtual storage.

target machine: The machine to which a program is being ported.

theoretical peak performance: The computation rate that can never be exceeded.

tune: To modify a program to promote better use of a machine's architecture and resources.

vector-enable: To modify a program to execute on a vector processor.

vector-enabling: Modifying a program to execute on a vector processor.

vector fraction: The fraction of a program that can execute on a vector processor.

vector length: The number of scalar data elements in a data vector.

vector machine instruction: A machine instruction for a vector processor.

vector processing: Executing a program on a vector processor.

vector section: A piece of a longer data vector.

vector section size: The maximum data vector length that a vector processor can operate on with a single vector instruction.

vector stride: The number of storage bytes between each scalar element of a data vector.

vector/scalar speed ratio: The ratio between calculation rates on a vector processor and its companion CPU.

VF: An IBM 3090 Vector Facility.

virtual storage: Storage that appears to exist in its entirety to a program, but in reality is simulated with a combination of real, expanded, and disk storage.

word: On the IBM 3090, a 4-byte quantity.

Vita

Alan Luchuk was born on September 9, 1962. He grew up in Tullahoma, Tennessee, and attended Bel-Aire Elementary from 1968-1974, West Junior High from 1974-1976, and Tullahoma High School from 1976-1980.

Alan enrolled at the University of Tennessee, Knoxville, in September 1980. He graduated with a Bachelor of Science degree in Electrical Engineering in August 1985. In September 1985, Alan again enrolled at the University of Tennessee, Knoxville. He graduated with a Master of Science degree in Electrical Engineering in August 1991.